(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
13 June 2002 (13.06.2002)

PCT

(10) International Publication Number
WO 02/46888 A2

(51) International Patent Classification[7]: G06F

(21) International Application Number: PCT/US01/51376

(22) International Filing Date:
6 November 2001 (06.11.2001)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
60/246,165 6 November 2000 (06.11.2000) US
60/246,185 6 November 2000 (06.11.2000) US

(71) Applicant: **BROADCOM CORPORATION** [US/US]; 16215 Alton Parkway, Irvine, CA 92618-3616 (US).

(72) Inventors: **NICKOLLS, John, R.**; 16215 Alton Parkway, Irvine, CA 92618-3616 (US). **MADAR, Lawrence, J., III**; 16215 Alton Parkway, Irvine, CA 92618-3616 (US). **JOHNSON, Scott, D.**; 16215 Alton Parkway, Irvine, CA 92618-3616 (US). **KAMBDUR, Kirthiranjan**; 16215 Alton Parkway, Irvine, CA 92618-3616 (US). **MIRSKY, Ethan**; 16215 Alton Parkway, Irvine, CA 92618-3616 (US). **MADAN, Barkha**; 16215 Alton Parkway, Irvine, CA 92618-3616 (US).

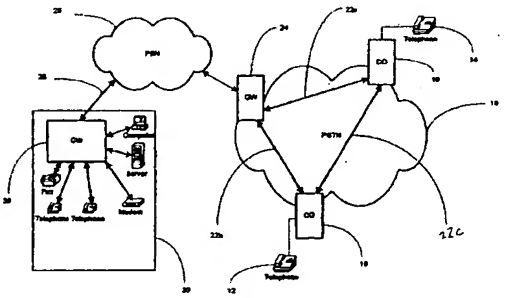
(74) Agent: **STRAWBRICH, Robert, C.**; Garlick, Harrison & Markison, L.L.P., P.O. Box 160727, Austin, TX 78716-0727 (US).

(84) Designated States *(regional)*: European patent (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE, TR).

Published:
— *without international search report and to be republished upon receipt of that report*

For two-letter codes and other abbreviations, refer to the "Guidance Notes on Codes and Abbreviations" appearing at the beginning of each regular issue of the PCT Gazette.

(54) Title: SHARED RESOURCE ARCHITECTURE FOR MULTICHANNEL PROCESSING SYSTEM



(57) Abstract: A signal processing system employs a shared resource architecture, sharing processor, memory and I/O resources to process cannelized data efficiently, with low power and a competitive cost per channel. The architecture employs a shared data memory between a plurality of processing engines, such that tasks may be reallocated among the processors based on capacity, or tasks may be worked on in parallel. The processing engines share a program memory that is independent of the data memory, which contains a single instance of all of the code required to process the incoming data. The program memory is only accessed on instruction cache misses in the engines. Both memories are shared over pipelined and arbitrated crossbar switches that service that service the requestors. The memories are broken into independent banks of memory so that multiple accesses may be serviced in parallel, except where there is contention. I/O is also shared among the engines, as the channelized data is routed to the appropriate assigned engines, and processed data is collected from the data memory. Supervisory processes also share the program memory and the data memory with the processing engines. The supervisory processors monitor the processing of the data, allocate the processing resources, and monitor the I/O process. The processing engines and their shared data memory may be duplicated as clusters, where the engines for each cluster share the cluster data memory with only those engines in the cluster, but share the program memory and I/O with all of the engines of all of the clusters. Interlocks are employed at the cluster level and at the system level to permit sharing of global and cluster resources in an orderly fashion. The architecture has particular application to voice-over packet (VOP) applications.

WO 02/46888 A2

SHARED RESOURCE ARCHITECTURE FOR MULTICHANNEL PROCESSING SYSTEM

CROSS-REFERENCE TO RELATED APPLICATIONS

This application claims the benefit of U.S. Provisional Applications Serial No. 60/246,185, Serial No. 60/246,239 and Serial No. 246,165, all filed November 6, 2000.

BACKGROUND OF THE INVENTION

Field of the Invention

This invention relates to the field of signal processing, and more particularly to multi-channel signal processing systems.

Description of the Related Art

At first, data communications were primarily transmitted over the public switched telephone network (PSTN) because it was already well established. While the PSTN had provided an existing network infrastructure for the emerging data communications market, the PSTN has never been the ideal solution for data transmission. First, the PSTN was originally designed to transmit and switch analog voice signals. Access to the PSTN for data transmission is often accomplished using existing analog voice lines, but only after the data is first converted into analog signals using a device such as a modem. Second, although the PSTN has been largely converted to a time division multiplexed (TDM) digital switch fabric, TDM is not the ideal choice for transmission of data that is bursty. Using TDM switching for a data channel requires that bandwidth over the channel is pre-allocated in an amount

sufficient to meet peak demand over that channel, even though the excess bandwidth may go unused much of the time for bursty types of data. Thus, while higher-speed transmission lines such as T1 lines have been used to access the digital switch fabric of the PSTN, this is not an ideal solution given the wasted bandwidth. Finally, as the demand for bandwidth allocated to data communications continues to increase, new network infrastructure must be built to meet the increased demand. It is often easier and less expensive to create networks dedicated to data transmission, such as packet-switched networks (PSNs) rather than add more capacity to the less than ideal PSTN.

Entities typically network their computers and servers together locally using well-known local area networks (LANS) such as Ethernet. These local networks may then be interconnected using a global or wide-area network such as a PSN, so that for example, remote facilities of the same business may communicate with one another or with other businesses connected to the PSN. The most common PSN is the Internet. Entities must also have voice communications facilities that serve the entity locally, as well as remotely. Often, an entity will employ a private branch exchange (PBX) to switch voice calls originated and received locally between voice equipment within the facility. The PBX has a connection to a proximate central office (CO) of the PSTN, by which a voice call originated or received within the entity's facility can be connected to other voice communication devices residing outside of the entity's private exchange. Voice communication devices typically include telephones, modems and faxes.

It is more expensive and less efficient for entities to maintain two separate networks, one for voice calls and one for data communications. Thus, it has been commonly recognized that to combine voice traffic over a PSN with data traffic would be highly advantageous. Figure 1 illustrates an example of a PSN 26 that is interfaced to the PSTN 10 through a gateway 24. Gateway 24 converts TDM voice traffic received from the PSTN 10 into data packets to be transmitted over the PSN 26 to the entity 30. GW 24 also converts voice data packets received from PSN 26 into TDM voice traffic to be switched by the PSTN 10. Gateway 36 can be employed to convert TDM voice traffic generated by the voice call equipment (e.g. telephones, faxes and modems) of entity 30 into data packets for transmission over PSN 26, and to convert voice data packets received from PSN 26 into TDM traffic destined for the voice equipment of the entity 30.

Thus, voice data originated by telephone 12 is received at CO 16 as analog signals, converted to digital TDM signals and then routed through the TDM switch fabric 22a-c in accordance with the destination of the call. If telephone 12 calls telephone 14, voice signals originating with phone 12 are routed to CO 18 over path 22c and CO 18 converts the TDM signals back to analog signals and the analog signals are transmitted to phone 14. Data originating at telephone 14 is transmitted back to telephone 12 in the opposite direction and is otherwise processed in the same way. If the call originated by telephone 12 is destined for a telephone residing within entity 30, the voice data will be routed through a gateway such as GW 24, which converts the TDM voice signals it receives to a packet data format such as Internet Protocol (IP) or asynchronous transfer mode (ATM), and sends the packetized voice data over PSN 26. In this way, voice traffic originating within or destined for entity 30 can be transported through a PSN as voice packet traffic along with packet data traffic. This voice-over-packet (VOP) arrangement can eliminate the requirement that entity 30 have a connection directly to a CO of PSTN 10 to address its voice communications requirements. Instead, the existing PSN 26 is used to satisfy both the voice communications requirements and data communications requirements of entity 30 without placing additional demand on the PSTN 10 infrastructure.

Gateways such as GW 24 heretofore have been quite costly and as a result not sufficiently cost-effective in competing with the PSTN. Converting TDM voice traffic into data packets and voice data packets back into TDM traffic requires a tremendous amount of real-time digital signal processing (DSP), and this processing must be performed with a high degree of accuracy, reliability and timeliness to provide the quality of service (QOS) requisite of voice communications. Figure 2 illustrates the stack of services that must be performed by the gateway 24 in both directions to process voice calls over a single channel. TDM gather/scatter and buffering block 80 represents the processes that are performed to convert TDM signals received at input 96 from the PSTN into 5 millisecond sample frames (gather) and to convert packets received from a PSN back into TDM frames (scatter) output over TDM out 98. Typically, the sample frames consist of 5 milliseconds of speech sampled at 8 KHz or 40 8-bit samples (i.e. 40 bytes). They also include header information that carries information regarding the origin and destination of the packet. TDM frames (which are different than sample frames) typically are made up of 24 channels of 1 byte samples at 8

KHz which equals 1.5 Mbps. Each TDM frame contains 24 slots each of which can be assigned to carry a single byte for each of 24 channels.

The packetization/jitter buffer block 90 outputs the processed samples over Packet Out 92 in the appropriate one of a number packet formats, including UDP, IP and RTP formats, as well as ATM formats such as AAL-1, AAL-2, AAL-5 etc. Packets received over Packet In 94 must be buffered using a jitter buffer to compensate for any packets that are dropped or delayed, and are then broken up into 5 ms samples. Echo cancellation block 82 represents the echo cancellation function that must be performed on each call. Tone detection block 86 monitors the type of call, and handles the processing required if the call turns out to be a modem or fax call, which includes emulating a fax or a modem on either end of the call so that the fax or modem equipment thinks it is communicating directly with the fax or modem at the other end. Vocoder block 88 implements any one of a number of compression/decompression algorithms, including G.711, G.729, G.728 and so on.

All of the functions of the service stack are well-known and are implemented by standard software that can be executed by a standard DSP chip. To be compliant, a gateway must be designed to handle all of the possible standards for I/O, compression/decompression, and to handle the different types of calls such as telephone, fax or modem calls, etc. because it never knows what kind of call it will receive or what kind of packet network to which it will be connected. Executing the requisite software to perform all of the functions of the services stack 100 for each packet of one channel in real-time requires a significant amount of computing power. This processing must be completed for each packet within the 5 ms sample frame. The amount of memory required to store the code for all of the variations of all of the functions, as well as data representing the state of each channel, is also significant. The I/O functions for each side of the gateway also presents significant resource requirements.

To be a viable solution, a gateway must be able to perform this processing for a significant number of voice channels. Implementing a system that provides the necessary computing power, memory and input/output (I/O) resources for what can be tens of thousands of voice channels, and doing so for a cost per channel that is competitive with the PSTN has been a most difficult challenge. Typical prior art gateway solutions employ the general architecture illustrated in Figure 3. A farm of commercially available DSP chips 50-56 are

employed to execute the various stack functions. Each DSP has a dedicated memory 58-64 respectively in which program code representing the functions to be executed is stored, as well as data representing the current state of the channel or channels being handled by the DSP. Separate I/O functions (not shown) coupled to the TDM lines 70 and the ATM/IP packet lines 72 are also replicated and dedicated to each DSP chip 50-56. Because the dedicated memory 58-64 for each DSP 50-56 was not large enough to hold all of the code representing all of the stack functions and their numerous variations, a bulk memory or external storage device 74 was shared among the DSP memories to contain the voluminous amounts of processing code for all of the stack functions.

To handle more channels for a given amount of resources, the processing resources would have to be leveraged to process a frame of data samples from more than one channel simultaneously. To accomplish this, channel data from multiple channels would have to be aggregated (or routed to the assigned DSP chip) through call aggregation block 66. The channels would be assigned using scheduling processor 68. Thus, if a particular DSP resource is assigned to four channels, and it does not have the processing power to handle a fourth call if it is one that is computationally intensive (e.g. a modem call), it would be desirable to reassign the channel to a DSP resource that is free to handle the call using a scheduling processor 68. But because the data memory is dedicated to each DSP, channel state data for the detected modem call is already stored in the dedicated memory of the originally assigned DSP and there is no easy and timely way to get the state data to the memory of the newly assigned DSP.

Because there is no way to predict what kind of voice calls will be made or any particular channel, and because it would be unacceptable to fail to handle a call because the assigned processor did not have the processing capacity available to process it, each processor can only be assigned to the number of channels that it can handle under the worst case scenario. As a result, processing capacity will go to waste because sufficient memory and processor resources must be allocated to meet the worst-case scenario, which may occur in as few as one percent of the calls over a given channel. If each DSP is only capable of handling one call when it is a modem call, then each DSP in this architecture can only be assigned one channel per DSP processor. This is true even though modem calls may only occur one percent of the time, thus rendering the processor resources underutilized as much

as 99 percent of the time. Employing the architecture of Figure 3 to handle only one voice channel (or even just a few channels) per DSP 54-56 is extremely expensive and therefore not cost-competitive with simple TDM circuits.

Therefore, it would be highly desirable to provide an improved multi-channel signal processing architecture that leverages computational, memory and I/O resources comparable to those employed in the prior art architecture of Figure 3 such that when it is applied to the conversion of voice traffic between TDM signals and packet signals, it is able to process several times the number of channels processed by the prior art architecture, thereby lowering the cost per channel to convert voice data between TDM signals and packet data signals. It would also be advantageous if the improved architecture could be easily implemented as a single integrated circuit. Finally, it is critical that the power dissipated per channel decrease in order to continue to meet power dissipation specifications for systems that would employ such an architecture.

SUMMARY OF THE INVENTION

BRIEF DESCRIPTION OF THE DRAWINGS

The multi-channel signal processing architecture of the invention may be better understood, and its numerous objectives, features and advantages made apparent to those skilled in the art by referencing the accompanying drawings. The use of the same reference number throughout the several figures designates a like or similar element.

Figure 1 illustrates the public-switched telephone network (PSTN) coupled to a packet-switched network (PSN) through a gateway.

Figure 2 illustrates generally the stack of services through which a voice call is processed by a VOP gateway.

Figure 3 illustrates a prior art architectural implementation of a signal processing system having resources able to perform the signal processing functions represented by the services stack of Fig. 2.

Figure 4 illustrates a block-level diagram of one embodiment of the multi-channel signal processing architecture of the invention as applied to a VOP gateway.

Figure 5.

Figure 6.

Figure 7.

Figure 8.

DETAILED DESCRIPTION

Overview

The multi-channel signal processing system architecture of the present invention facilitates the sharing of processing, memory and I/O resources across all of the channels the system is processing. In this application, the term data channel is intended generally to refer to any partition of data for whatever purpose, including the partitioning of data based on a particular source and destination, unless otherwise specifically limited. This sharing of pooled processing and data memory resources permits the resources to be dynamically allocated across all of the channels in response to variations in the demand for the resource pool at any given time. The sharing of program memory among the processing resources enables the system to operate with only one instance of code for each function that might be executed to process channel data. I/O resources are also shared to further reduce the amount of circuitry required to support the signal processing of multiple channels. Reducing the program memory size as well as the number of I/O circuits facilitates the implementation of the signal processing system of the invention on a single integrated circuit and reduces the amount of power dissipated per channel.

As specifically applied to a gateway providing VOP services, the invention is able to meet the variation in types of voice calls with complete flexibility such that any service is available at any channel processed by the architecture. The invention is able to reallocate channel assignments dynamically for every 5 to 10 ms data cell or sample frame processed by the system to meet the changing processing demands of the channels as a whole. Thus, if an increase in processing demand occurs due to the sudden influx of modem or fax calls for

example, the processing resources of the system can be reassigned and brought to bear on the problem from the total pool of such resources. In this way, the total pool of processing, memory and I/O resources can be determined based on a statistical model of the aggregate plurality of voice calls that can be expected at any given time rather than the worst case demand per channel. The multi-channel system architecture brings together a large number of channels to be processed into a system of shared resources (e.g. memories, processors, and IO interfaces) so that the large number of channels can be statistically leveraged to consume fewer resources than required by prior art systems. The multi-channel signal processing system architecture of the present invention as applied to VOP permits many more channels to be processed for a given set of resources than was possible based on prior art system architectures, thereby significantly lowering the cost per channel processed. Moreover, the system architecture of the present invention facilitates its implementation on a single integrated circuit, further lowering the cost of implementation. Finally, the average power dissipation per channel is also significantly reduced, which is crucial for meeting established power dissipation specifications.

In one embodiment of the system architecture of the invention, the signal processing and I/O resources share a common data memory in which the sample data, packet data, and state for each channel are maintained. The data memory is shared among I/O and processing resources over a pipelined crossbar switch. The data memory is partitioned into several banks that can be independently accessed so that multiple requests for access by the pooled resources may be serviced per clock cycle. Resource access to the data memory is controlled by an arbiter that handles overlapping requests for access to the same portion of the memory by the pooled processing resources and the I/O.

This pooling of processing, I/O and data memory resources is sometimes referred to herein as a cluster, and the data memory is sometimes referred to herein as a cluster memory. In one embodiment, the resources of the cluster are limited to thereby limit the physical implementation area required and thus enable signals to rapidly propagate within the cluster. The processor resources of the cluster includes a plurality of packet and signal processing engines sometimes referred to herein as cluster engines, processing engines or vector reconfigurable engines. The processor resources of the cluster may also include a scheduling processor, sometimes referred to herein as a cluster processor, which runs an operating

system and dynamically assigns the processing engines of the cluster to particular channels as often as once per data cell or sample frame to ensure real-time completion of the digital signal processing functions on each data cell or sample frame for each channel as dictated by the services stack.

The I/O and processing resources of the cluster also share a global program and data memory in which a single instance of executable code for every one of the standard signal processing functions of the services stack resides. Each of the cluster processors and processing engines includes an instruction cache that is filled from the shared memory each time a cluster processor or processing engine experiences an instruction cache miss. In one embodiment, a plurality of clusters can share the program memory and the I/O resources. The program memory is shared using an arbitrated and pipelined crossbar switch similar to the one used by the processing resources of the cluster to share the cluster (or data) memory. The shared program memory also contains a high level operating system executed by a main processor to coordinate the sharing of resources among the clusters. The shared program memory also stores the current state of the system, including the channels to which each of the clusters has been assigned and the location in memory at which the incoming data and outgoing data for the channel are stored in the data memory of the cluster.

One or more TDM and network ports are shared among the clusters through the cluster I/O resources. The TDM port receives TDM framed data from TDM inputs and performs the function of obtaining each byte of TDM data and scattering it to its channel buffer and converting it into sample frames. The scattered samples for each channel are directed to the cluster and the specific location of an input buffer in the cluster memory associated with the processing engine currently assigned to handle the channel from which the data was sourced. The TDM port also receives processed data samples from the cluster memories destined for TDM transmission and gathers the data samples into the TDM frame format for output to TDM destinations such as the PSTN. Likewise, one or more network cell bus ports receive packet data from a packet data source such as a PSN and direct the data cells to the input channel buffer associated with the cluster processing engine assigned to handle the channel over which the data cell was received. The network cell bus port also takes data cells processed by the system and outputs them to packet switched destinations such as the PSN.

Employing the multi-channel system architecture of the present invention to perform the digital signal processing for a VOP gateway application as described, several hundred channels of voice data may be processed by a total pooling of processing, memory and I/O resources that can fit into one integrated circuit and which as a result is highly cost-competitive with prior VOP systems employing farms of DSP devices. With reference to Figure 4, one embodiment of the multi-channel system architecture as applied to the VOP gateway application is described. In a typical VOP application, the clusters 100, 200, 300, 400 are dedicated to performing the bulk of the computation required. The cluster engines 112-118, 212-218, 312–318 and 412-418 primarily handle the heavy-duty signal processing and packet processing tasks, while the cluster processors 102, 202, 302 and 402 handle higher-level protocols and management functions. They also coordinate with other clusters and a main processor (located in the master cluster 500) through the global shared memory 508, 510 and their respective cluster synchronization hubs 108, 208, 308 and 408. The I/O units (including an I/O arbiter 506, two line-side TDM serial ports 514, two network cell bus serial ports 512, a processor host port interface 518, and a serial boot port 516) autonomously move input data into the cluster memories 101, 201, 301, and 401 and shared memories 508, 510, and take output data from the same memories and move it out of the chip. This process is coordinated by the operating system (OS), stored in the shared memories 508, 510 and running on the main processor and cluster processors using global control registers which are part of the global resources 504. The OS also communicates with the host system (not shown), either through a host port 518 or the network cell bus port 512. The global on-chip shared memory 508, 510 is used to hold frequently used code and data that is shared between clusters 100, 200, 300, 400. Cluster-specific code and data is typically kept in the cluster memory. The external memory (DRAM) 522, when present, is used to hold large quantities of less frequently used data or program code that is accessed directly or paged onto the chip when needed.

The Main Cluster

In one embodiment, the system has a main cluster 500, which contains a RISC processor (the main processor or MP 600), its bus interface (BIF) 606 and instruction 602 and data 604 caches, a DMA engine (memory bridge 614), and several local registers 608. Figure 5 shows a block diagram of the main cluster 500. The main cluster's primary job is system

management. It can execute programs from the external memory 522 (Fig. 4) or on-chip shared memory 508, 510. It can access the global resources 504 and synchronization hub in order to control the rest of the chip. The main processor 600 controls the host port interface 518. The memory bridge 614 can transfer data blocks between the two shared memory resources (i.e. external DRAM 522 and on-chip SRAM banks 508, 510). The main cluster 500 can share data with the four processing clusters 100, 200, 300, 400 via the shared memory banks 508, 510. Control communication between the main cluster 500 and the processing clusters occurs through messages in the shared memory banks 508, 510, and signals in the global resources 504 (i.e. locks, barriers, or interrupts).

In one embodiment, the main processor (MP) 600 is a RISC processor core configured for the system. The MP 600 manages a 4 Kbyte instruction cache 602 and a 4 Kbyte data cache 604. In one embodiment, both can be direct-mapped and write-through. The MP 600 communicates with the rest of the system through its bus interface (BIF) 606. The BIF 606 connects the main processor 600 to the rest of the system. Through the BIF 606, MP 600 can access the shared memory banks 508, 510 and external memory 522, the memory bridge 614, the global resources 504 that include a synchronization hub and I/O port control registers, the host port interface 518 and its local registers 608. The main processor 600 presents a read or write request to the BIF 606. Based on the address BIF 606 determines the resource for which the request is destined. Once the BIF 600 has determined which resource the request requires, it generates the appropriate transaction sequence to that resource. If the request is a read, the BIF 606 passes the return data to the MP 600.

In addition to routing read/write requests, the BIF 606 performs a number of vital functions for the MP 600. One is memory protection. The BIF 606 compares the processor read/write request address to a set of OS-controlled bounds registers and disallows any illegal request. A second is write gathering. For write transactions to the shared memory 508, 510, BIF 606 gathers write requests, sending data out to the memory only when necessary. Write gathering buffers reduce the amount of traffic at the global shared memory arbiter 502. A third function performed by BIF 606 is memory operation order guarantee. BIF 606 guarantees that certain memory transaction sequences will complete in order. It stalls MP 600 as necessary to accomplish this.

The main processor 600 has a number of local registers 608 it uses to control itself and its immediate environment. These include bounds registers for memory protection and a fault register to capture offending addresses. A control to force a global "invalidate" on either the instruction 602 or data caches 604. The processor's ID number, so that the OS can distinguish the MP 600 from the cluster processors (102, 202, 302, 402 Fig. 4). Finally, special registers that cause the MP 600 to stall until certain transactions are complete to ensure memory operation order guarantee.

Main processor 600's memory bridge 614 is a DMA engine that can move data to and from anywhere in the global shared memory (i.e. external DRAM 522 and on-chip shared memory 508, 510) to anywhere else in the global shared memory. To initiate a transfer, MP 600 writes a command into control registers of memory bridge 614. The command may consist of source and destination start addresses and a length. In one embodiment, memory bridge 614 performs transfers in multiples of 16 bytes aligned to 16-byte boundaries.

Memory bridge 614 does not perform any address manipulation (such as circular addressing or striding). However, it may perform a block parity function on the data being transferred. The resulting block parity word may be read by main processor 600 or compared against an intended value supplied by MP 600. This can be used to check data for errors caused by single-bit errors in DRAM. The parity check function in the memory bridge 614 can be used even when no data is being moved. Memory bridge 614 supports an audit mode in which a DMA is setup normally and the data is read, but is never written anywhere. Rather, it is read solely for the purpose of performing the parity check, usually on the external DRAM.

I/O bridge 612 links an I/O bus and the global shared memory 508, 510. It takes requests generated by the I/O arbiter 610 and generates the appropriate reads or writes to the shared memory 508, 510 through shared memory arbiter 502. I/O bridge 612 operates independently from the rest of main cluster 500 and supports I/O transfers to the shared memory 508, 510 in the same way as cluster I/O bridges 106, 206, 306 and 406 handle I/O to and from their respective cluster memories 101, 201, 301, 401 (Fig. 4) as described below.

Global Resources

In one embodiment, global resources (504, Fig. 4) provide mechanisms for the system's processors to coordinate their efforts and control the operation of the chip. Figure 7 provides a block-level description of the global resources 504. In one embodiment of the system, the global resources provide a number of control mechanisms to wit: locks through lock registers 624, barriers 626, interrupts through global control registers 628 and an I/O control path 630. The global resources 504 are linked directly to each of the system's control processors 102, 202, 302, 402, 600. The global resources 504 include an arbiter 632 that grants one global resource transaction per cycle.

In one embodiment, the lock mechanism provides a means of protecting shared resources and memory. It also ensures that only one processor can access them at a time. The locks consist of 16 eight-bit lock registers 624 that can be set to a value only if they are currently free (i.e. they contain zero, which is the free code). Whenever they hold a non-zero value, only a clear operation (i.e. writes of zero to the register) will have an effect. Assignment of the locks is based on software conventions and is therefore up to the OS to determine how they are used and assigned. For example, the OS can declare that one lock register is used to secure a block of memory for example. By software convention, in order to write to that memory a processor must first set the assigned lock register to its unique processor ID. The processor must then read the register to confirm that it has been set to its ID. If the processor has successfully written its ID to the register, it has successfully acquired exclusive access to the resource. If another ID is found in the register, the processor must wait and try again to write its ID. This process continues until the processor succeeds. Thus, the lock mechanism ensures that the set operation will only succeed if no other processor currently has acquired the resource. The processors must explicitly free the lock after they no longer need the resource being protected by writing zero to it.

Barrier mechanism 626 provides a means for the processors to synchronize themselves. Barriers 626 consist of a barrier configuration register (not shown), an eight-bit input value register (not shown) and an eight-bit return value for each processor(not shown). The barrier config registers allow the processors to be configurably grouped as participants and non-participants in the current barrier. The value registers allow the processors to communicate a small amount of information along with the barrier. As with the locks, the use of the barriers is based on software conventions. Typically, the processors set up the

barrier registers to group themselves as participants or non-participants in the current barrier. Then each of the participating processors proceed with their computation. When a processor reaches the barrier point in its code, it sets a value in the value register. When all the participating processors have arrived at the barrier and set values in the value registers, the values are bit-wise ANDed and then presented back to the participating processors along with a flag indicating that the barrier has been reached. The processors may poll the barrier logic to determine if the flag has been set.

The global registers 628 are a small set of read/write registers that are used to control the entire chip. One global register is the processor on/off configuration register, one for each processor, which allow the OS to turn a processor completely off if it is not needed. These registers are also used by the boot mechanism to turn on a processor during the boot-up process. The global registers 628 further include five non-maskable interrupt registers, one for each processor, and five maskable interrupt registers. These registers allow the OS in any CP 102, 202, 302, 402 or MP 600 to forcibly interrupt that CP or MP if necessary.

The Cluster.

In one embodiment, the system's clusters perform the bulk of the signal processing work that the system is tasked to provide. A block diagram of a cluster is illustrated in Figure 8. The cluster processor block 802 is very similar to the main cluster 500 previously described. A block diagram of the cluster processor block, including associated components is illustrated in Figure 9. It includes a RISC processor 826 having local instruction 824 and data caches 822, a bus interface (BIF) unit 828, and local registers 830. The primary function of the cluster processor block 802 is to manage the operation of the cluster 800. Cluster processor block 802 is responsible for scheduling tasks on the engines (812, 814, 816, 818 Fig. 8) and monitoring their progress. It is also responsible for controlling data transfers between the shared memory system (Fig. 6) and the cluster memory (801, Fig. 8) using the cluster memory bridge 804. In addition, CP 802 may also perform packet and signal processing work in support of the engines 812-818.

In one embodiment, the CP 826 is a RISC processor similar to the main processor 600. It manages a 4 Kbyte instruction cache 824 and a 4 Kbyte data cache 822 and communicates with the rest of the chip through its bus interface (BIF) 828. Through BIF

828, CP 802 can access the global shared memory (i.e. shared 508, 510 and external memory 522), the global resources 630 (including the locks and barriers), the cluster memory bridge 804, the cluster memory 801, the cluster resources and engine debug registers (not shown), and its local registers 608. BIF 828 of CP 802 performs the same functionality as BIF 606 of the master cluster 500, including transaction routing, memory protection, write gathering and execution order guarantees. For the CP BIF 828, write gathering is also performed for cluster memory accesses. In one embodiment, CP BIF 828 has some additional functionality that is required to support the cluster memory 801. Because cluster memory accesses are only 32-bits wide, during a cache line read for CP 802, BIF 828 can issue four separate transactions and gather the results.

Local register set 830 of CP 802 serves the same functionality as local register set 608 for master cluster 500, including memory protection bounds, global cache invalidate, processor ID and transaction stall flags.

Local register set 830 of CP 802 serves the same functionality as local register set 608 for master cluster 500, including memory protection bounds, global cache invalidate, processor ID and transaction stall flags.

The processing engines 812, 814, 816, 818 of cluster 800 are responsible for performing the bulk of the signal processing work, and when applied to VOP, the data-intensive packet processing work that the system is called upon to perform as previously described. In one embodiment, there are four engines in each cluster, and 16 in a system having four clusters (Fig. 4). One embodiment of an engine 812-814 is illustrated as a block diagram in Figure 10. In one embodiment, the engine contains two major datapaths - a reconfigurable vector datapath and a scalar datapath. The vector datapath consists of a 1024 byte register file 850 that is addressable in 1, 2, and 4 byte words, four computation units, two 40-bit accumulate registers 852a, 852b, and several addressing units. Its compute units include a 16 x 16 multiplier 854, a 16/32/40 bit ALU 856, a variable shift unit 858 and a specialized bit- operation unit 860. The vector datapath is designed to run semi-autonomously based on the configuration stored in configuration registers located in block 862 as controlled by instructions. The scalar datapath consists of a smaller 16 x 32 bit register file 864 and a 32 bit ALU 866. This datapath runs fully under control of the

instruction stream. It is primarily intended to run C code that is needed between and within signal processing loops.

The instruction memory and configuration registers 862 controls the operation of the engine. In one embodiment, the instruction memory is managed as an instruction cache. The configuration registers are used to expand the semantic power of the instruction, allowing complex operations to be easily controlled and executed, without requiring a great deal of special-purpose hardware. The engine's instruction memory and configuration registers 862 are loaded directly from cluster memory 801 or global shared memory (508, 510, 522) by the cluster's memory bridge 804 upon requests from the engine 812,814, 816, 818. The engines address cluster memory 801 through the memory address unit (MAU) 868, which transfers data to/from the vector file 850 and the scalar register file864. The engines cannot access global shared memory (except for instruction and configuration loads). Because engine 812-818 is a synchronous system, it stalls when it encounters contention during a cluster memory 801 reference.

Engines 812-818 may also access the cluster resources for mutual exclusion locks and barrier synchronization functions. In addition, engine 812-818 can generate interrupts for CP 802 through the cluster resources. The cluster processor 802 uses the cluster resources path to access the internal state of engine 812-818 for debug and control purposes. Finally, engine 812-818 can be linked with the other engines in the cluster in a configurable ring interconnect. Each engine can send one 32 bit word per cycle directly to a neighboring engine and can be used to perform multi-engine systolic and data parallel computation.

Engine 812-818 serves to reduce the number of accesses to both shared program memory 508, 510 and shared cluster data memory through use of its instruction memory or

structured to facilitate simultaneous accesses by multiple requestors with minimal contention. It is also pipelined to be sure that requests are acknowledged and processed on every clock cycle. A block diagram representation for one embodiment of the cluster memory is shown in Figure 11. Cluster memory 801 consists of eight 32-Kbyte banks of SRAM 801a-h, each with an independent port to the cluster arbiter 810. Each bank can perform a single 32 bit read or write per cycle. The cluster arbiter 810 accepts requests from the seven possible requestors and arbitrates (in a round-robin fashion) independently for each bank. With no contention, each requestor can make one access per cycle, or in other words the memory can process up to seven requests per cycle. In one embodiment, the memory banks are interleaved every 32-bit word. This causes sequential accesses (the most common type) to hit a bank only once every 8 cycles, further limiting contention.

The cluster memory arbiter 810 provides status to the requestors indicating when their access is granted. This allows the requestors, particularly the engines 812-818, to stall while their request is pending. Because many of the I/O protocols do not permit stalling, the I/O bridge can be always granted priority. In one embodiment, the requestors and memory banks are interconnected through a crossbar switch that cross-connects all of the banks with all of the requestors 870. This embodiment has been shown to sustain an aggregate bandwidth for four clusters of up to 4.6GB/s of read/write traffic at a clock rate of 166MHz.

A block diagram of one embodiment of the full CM 801 structure is shown in Figure 13. The four major elements which make up the full CM 801 are the eight 32 KByte x 32 bit single access, byte enabled physical memories 801a-h, eight independent arbiters / transaction source multiplexers 809a-h, a bus-based crossbar switch interconnect network 810a, and seven data return multiplexers and their associated control logic 811a-g. A detailed diagram of one embodiment of the return multiplexers 811a-g is shown in Figure 14a. A more detailed diagram of one embodiment of the eight independent arbiters / transaction source multiplexers 809a-h is shown in Figure 14b. A more detailed description of one embodiment of the requestor front-end interface (REFI) of the eight independent arbiters / transaction source multiplexers 809a-h of Figs. 13 and 14b is illustrated in Fig. 14c. For a more detailed description of embodiments of the cluster memory system, see the patent application entitled "Pipelined Multi-Access Memory," filed November 2, 2001 and which is incorporated herein in its entirety by this reference.

The cluster memory bridge 804 links the cluster memory 801 and global shared memory 508, 510, 522. A block diagram of one embodiment of the cluster memory bridge 804 is shown in Figure 12. It serves two primary functions. First, it moves blocks of data between cluster memory 801 and shared memory 508, 510, 522 (and between addresses within both regions) under the command of the cluster processor 802 through block transfer engine 880. This function is virtually identical to the one performed by the MP memory bridge (614, Fig. 5), with the addition of the cluster memory interface.

The second function is to fetch instruction pages for the engines 812-818 and configurations from shared memory 508, 510 or cluster memory 801. These operations are performed upon requests (e.g. cache misses and instructions to load instruction pages or load configurations) from the engines 812-818 and proceed independently from the block transfer engine 880. Because instruction pages (I-pages) and configurations have fixed block sizes, the instruction page engine 804 automatically generates the sequence of addresses required to fetch the specified page or configuration. One embodiment of the bridge has a 128 bit read/32 bit write port into the global shared memory. It prioritizes its requests with configurations first, followed by I-pages and finally by block transfers. Like the MP Bridge, the cluster memory bridge can perform a parity check on the data it transfers. This includes block transfer and engine I-pages, and configurations.

The cluster I/O bridge (806, Fig. 8) is a simple mechanism that links the I/O bus 820 to the cluster memory. The I/O bridge 806 handles the bus and memory protocols and moves the data between them. All status information on the I/O transfer is written into the cluster memory as part of the I/ O data.

Cluster resources serve two major functions. One is to provide a synchronization hub for engines and a communication path between the CP 802 and engines 812-818. The cluster sync hub 808 performs virtually the same function as the global sync hub discussed earlier. It provides 16 locks and a configurable fast barrier sync mechanism for the CP 802 and engines 812-818. In one embodiment, the CP 802 and engines 812-818 are all directly coupled with the cluster synchronization hub 808. The hub 808 provides mutual exclusion locks, barrier synchronization, and multi-processor bitwise aggregate functions for CP and/or engines cooperating in parallel in the cluster.

Each processor is numbered: the cluster processor (CP) is numbered 0 and the Engines are numbered 1 through 4, within their cluster. Each processor can read its processor number from an IPROC register – the Engines use the "riproc" instruction to read their cluster number and processor number within the cluster.

A set of cooperating processors (any of the engines and the CP) within the same cluster can work in parallel on the same task in parallel fashion. The set of cooperating processors each perform their part of the task and share or exchange results after synchronizing at a barrier to wait for their peers to finish their parts. Inter-processor communication may be via the cluster memory, bitwise aggregate operations, or the inter-engine nearest neighbor interconnection network.

The barrier synchronization hub 808 permits multi-processor parallel programs to efficiently guarantee correct ordering of operations. When each processor in the cluster arrives at a barrier, it waits until all of the participating processors have arrived, and only then does it continue executing its code. This guarantees that the code after a barrier is only executed after *all* the processors have finished executing the code before the barrier. The barrier synchronization hub 808 is dynamically reconfigurable to support different concurrent barrier sets needed by tasks using multiple processors in a coordinated parallel fashion. Each processor has its own set of barrier registers in the hub to support reconfigurable barrier sets. Processors write their waiting flag in the hub to indicate they have arrived at the barrier. The hub uses a simple AND function to detect when all processors have arrived, and sets a done flag for each processor waiting at that barrier. The barrier waiting and done flags are interlocked so that when one sets, the other is reset, so that complex acknowledgements are not needed.

Software barriers using the cluster memory would require many clock cycles, with time proportional to the number of cooperating processors. The barrier synchronization hub completes a barrier just one clock after the last processor arrives at the barrier, enabling fast fine-grain synchronization. A bitwise 8-bit aggregate AND function is also computed among the participating processors at each barrier and returned to all processors when the barrier is done. In parallel algorithms, program control flow is often determined by answering questions like "have all processors solved their sub-problems?" or "does any processor need to execute the following operation?" The *all* voting operation detects if every processor has

voted true for a given decision, and is the aggregate AND function. The *any* voting operation allows a processor to detect if any of the processors have voted true, and is obtained via complementing the inputs and outputs of the aggregate AND function. Each CP and Engine processor is connected to its own set of barrier registers in the cluster synchronization hub.

Figure 15b shows the barrier hub registers for a processor[i] 900, where i ranges from 0 to 4. The set of participating processors is established in register bar_set[i] 904 using a software instruction. Processor[j] is included in processor[i]'s barrier set if bit j of bar_set[i] 904 is set to 1 and bit i of bar_set[j] (not shown) is set to 1. Each processor has its own bar_set[i] register 904 to permit software to dynamically change the set of participating processors as determined by a software process on that processor. Processor[i] 900 waits for those processors that are in its set and which also have processor[i] 900 in their set. The interlocked sets enable the set of participating processors to be increased and reduced as needed.

Processor[i] 900 arrives at a barrier 813 by writing a 1-bit bar_wait[i] 90b flag to 1, which the engine does with either a putbar or waitbar instruction. It simultaneously writes an 8-bit bar_arg[i] 908 value, and resets the previous bar_done[i] 910 flag to 0. When all the processors in the barrier set arrive at the barrier, the bar_done[i] flag 910 is set in each participating processor, the bar_wait[i] flag 906 is reset, and a bar_and[i] register 910 gets the aggregate AND of the 8-bit bar_arg[j] values from each participating processor[j]. The 8-bit bar_and[i] 910 value and the 1-bit bar_done[i] status 910 are returned to the processor, via the getbar or waitbar engine instruction.

The cluster processor and engines share an array of lock registers 815 in the cluster synchronization hub 808. The lock registers 815 are used by operating system software to provide mutually exclusive access to objects or resources shared by multiple processors. The cluster memory 801 holds software objects that are shared between engines and the CP. The lock registers perform an atomic read-modify-write operation when written, so that an ordinary CP store can serve as the access mechanism. The lock registers are provided because one embodiment of the system's cluster processor implementation lacks an atomic read-modify-write operation on memory, such as test-and-set or compare-and-swap. The lock mechanism is simpler and requires less logic than a shared memory system providing atomic read-modify-write operations on every location in memory. The array of 16 lock

registers is 8-bits wide. By software convention, each lock register address is associated with a particular object or resource. The identification of the owner is stored in the 8-bit lock value. A zero value indicates the lock is free, there is no owner.

Access is gained in the same way as for the locks previously discussed with respect to the global resources (504, Fig. 4). An engine or the CP may attempt to obtain exclusive ownership of the lock by writing its non-zero 8-bit ID to the lock register. The write is permitted only if the lock ID is already 0. An engine or the CP may release a lock by writing the value 0 to the lock register, which always succeeds. The CP or any engine may release any lock; there is no check for the true owner.

A CP to engine connection is used by the engines to generate interrupts for the CP. An engine does this under software control, typically when it encounters a problem that it cannot handle or when it runs out of work to do. The CP uses the connection to read and write the engine's state registers, primarily for debug purposes.

The Shared I/O

The system's I/O ports connect the chip with the outside world. There are five different kinds of I/O ports, each with a dedicated purpose. Figure 18 shows a block diagram of one embodiment of the invention with the four main I/O interfaces. The fifth port, General Purpose I/O, connects to a different system and is not shown here. Each of the I/O interfaces connects to the on-chip memory through an I/O Arbiter 506 and two 32-bit shared buses. One of the I/O buses is handles incoming data, the other is for outgoing data. These buses are shown together as 595, 597 in Figure 17 for simplicity. They are illustrated separately in Fig. 18. Each I/O port is responsible for management of the external I/O protocol as well as generating both read and write transactions over the I/O bus. Control registers in the I/O interfaces, other than the Host Port 518, are accessed by the processors through the global resources. Host Port control registers are accessed directly from the MP's BIF.

The I/O Arbiter 506 grants use of the I/O buses in a round robin fashion. Because many of the I/O interfaces are not permitted to stall, the arbiter 506 ensures that requests are serviced within strict latency limits. When the arbiter grants access to the bus for incoming data, the interface writes the data to the bus, along with an address and tag indicating its destination. The write data is broadcast to all of the clusters, so the tag can specify one or

more of the cluster/ shared memories. This feature is particularly useful for circumstances where, for example, the destination of the packet is not yet known or where more than one channel across a number of clusters is to receive broadcast information such as music while on hold or multi-party conferencing.

The individual I/O Bridges receive the data and tag and write to their respective cluster memories if the tag matches. For outgoing data, the interface passes the address and tag requesting the data. The appropriate I/O Bridge performs the read and returns the data to the interface over the return I/O bus. The system's I/O Bridge 612 subsystem provides the data path between the system and the external interfaces.

The I/O bridge subsystem includes the following I/O ports: two TDM ports 514, two network cell bus ports 512, the Host Port 518, and the Serial Boot Port 516. This subsystem enables the I/O ports to initiate read/write transactions for data movement to or from the cluster memories and shared RAM to the outside world. Subsystem operation is by a DMA-style model of data movement. The I/O Bridge subsystem consists of three main components working together. The first major component, the I/O arbiter, serializes transactions from the ten I/O port requestors and broadcasts them onto a single interconnection bus, which is called the "Runway Bus" (597, Fig. 18). The Runway Bus 597 carries transaction requests and addresses to the four clusters. Inside each cluster, the Runway Bus terminates at the third major component, the Cluster I/O Bridge 106, 206, 306, 406. This bridge accepts transaction requests, passes them to the cluster memory arbiter for access to the cluster memory, and returns read data on the Runway Bus. This subsystem can also access shared SRAM, using the very same mechanism. The I/O arbiter 506 sends transactions to the shared SRAM 508, 510 exactly as though it were a fifth cluster. These transactions are received by a block called the Shared SRAM I/O Bridge (1000, Fig. 19), which is a design variant of a cluster I/O bridge. The Shared SRAM I/O Bridge 1000 passes the transactions to the shared SRAM arbiter for access to the memory.

As illustrated by Figure 19, the I/O arbiter 506 has three broadcast ports out of which it sends transactions: one port 199 serves clusters 100, 300, one port 197 serves clusters 200, 400, and one 195 serves the shared SRAM through the shared SRAM I/O bridge. Through these ports, transaction requests (address, byte enables, cluster ID, ReadNotWrite, and write

data) are delivered once per cycle. Read data is returned on a separate bus, which is also shown in the figure as 189, 187 and 185 respectively.

Two principles of operation are used by this subsystem: a new transaction request may be dispatched every clock period, and the latency for read data return is fixed. The ability to dispatch a new transaction every clock period not only creates deep pipelining, but also has the effect of propagating some I/O requirements deeper into the system. Specifically, the implementation of the network cell bus 512 and line-side TDM 514 ports do not have hardware flow control. The protocols used for these ports expect that a byte will be available for transmission, or that a received byte will be accepted, in its particular time slot, every time. This expectation is carried into the Runway Bus 597 in the form of an open-loop broadcast of transaction requests without a mechanism for the receiver to indicate that it is busy or not ready. This in turn propagates into the clusters (and shared SRAM arbiter) as a requirement that I/O transactions be served with a deterministic maximum latency, which in turn requires that I/O be given priority.

The fixed latency for returning read data operation is also a direct consequence of the latency requirements at the external ports. When a read transaction is broadcast on the Runway Bus 597, read data is expected back a fixed number of clocks later. The only exception to the fixed latency is that requests to the shared SRAM may have a different fixed latency than requests to the cluster SRAM's. However, it is still known at the time a transaction is launched exactly when the read data will be returned.

The protocol by which an I/O port passes a transaction request to the I/O arbiter is designed to be as simple as possible. On a given clock edge when the I/O port wishes to initiate a transaction, it indicates this by driving to the I/O arbiter a read request and write request signal, only one of which may be active. At the same time, the I/O port drives an address, a cluster ID, byte enables, and valid write data if the request is for a write. The I/O port continues driving these values until a clock edge has occurred on which the I/O arbiter was returning an active "ready" signal while the request was being driven. As soon as the "ready" is returned, the transaction is complete from the point of view of the I/O port. If the transaction was a write, it has been accepted (posted). If the transaction was a read, read data was returned at the same time as the "ready" signal (the I/O port may use the "ready" signal to enable its receive-data register). The I/O port may then drive a new transaction request.

This protocol allows the I/O arbiter to accept and FIFO any number of write transactions without the I/O port's knowledge. However, read transactions are not pipelined. The I/O port waits until read data is returned before it launches another transaction.

When the I/O Arbiter 506 accepts a transaction from one of the I/O ports, it sends it to the cluster(s) using the Runway Bus 597. The Runway Bus 597 is an open-loop unidirectional broadcast of transaction requests, which may occur as frequently as every clock period. There is no ready return or acknowledge; the receivers are expected to handle transactions as often as they can occur. The I/O Arbiter 506 dispatches a read transaction by driving the request signal high to indicate a request while driving the read/write signal to indicate a read. A read address and address-related information (byte enables, cluster ID) are driven at the same time. On the next clock period, the transaction is implicitly accepted. A fixed number of cycles after that, read data will be returned from the cluster on the read data bus. There is no "valid" bit to indicate that this has occurred, so the I/O Arbiter counts clock periods.

The latency for the return data is pre-defined and fixed, with one sole exception: it may be permitted to be one value for the four clusters and a larger value for the shared SRAM. In this case, there is a potential drive conflict on the return data bus whenever the shared SRAM may return data on the same cycle as another cluster. The I/O Arbiter may handle this in either of two different ways. It may accept simultaneous return data from the shared SRAM and from another cluster on two separate sets of wires and return the read data to two requestors on the same cycle. Alternatively, the I/O Arbiter may use its knowledge of the return data latency to detect, *a priori*, that the collision will occur, and prevent it by holding off dispatching the second transaction. For example, if the shared SRAM latency is 9 cycles and the cluster memory latency is 7 cycles, the I/O arbiter could prevent any collisions simply by refraining from launching a read request to a cluster memory two cycles after it launched a read request to the shared SRAM. It could launch a different transaction in that cycle or it could just stall one clock period, depending on the availability of requests and the sophistication of the I/O Arbiter.

The write transaction is indicated by transmit request with read/write set to write. The transaction is implicitly accepted on the next clock, after which the I/O arbiter considers it complete.

The I/O arbiter may accept return data from different clusters on separate wires. Referring to Figure 19, the I/O Arbiter 506 has three separate ports 199, 197 and 195. Each is a copy of the Runway Bus 597, with its own transaction-broadcast path and its own read-data return path, servicing two clusters. There are four identical Cluster I/O Bridges in the system, one in each cluster. Their job is to accept transactions on the Runway Bus as shown in Figure 3.1, and handle them at a sustained rate of one new transaction per clock. The primary function of a Cluster I/O Bridge is to translate transactions on the Runway Bus protocol to transactions that can be accepted by the cluster SRAM arbiter. Because the Runway Bus and cluster SRAM arbiter protocols are almost identical, this operation is a matter of latching the transaction and sending it along.

The Shared SRAM I/O Bridge performs the same job as the Cluster I/O Bridge, receiving transaction requests from the Runway Bus and passing them to a memory arbiter for completion. This block is a design variant of the Cluster I/O Bridge. Just as was the case in the cluster I/O bridge, the shared SRAM arbiter is required to give priority and a guaranteed maximum latency to I/O requests. Even so, the shared SRAM memory subsystem may suffer a longer latency than the cluster memory subsystem. Therefore, the shared SRAM I/O bridge will have a deeper pipeline, and will have a longer (but still fixed) latency for returning read data.

The serial boot port 516 is designed to give the system the ability to initiate its own boot-up by fetching a byte stream from a standard serial EEPROM in the system and copying it into the system's internal shared SRAM 508, 510. The port performs this function without glue logic and without the intervention of an external CPU. The protocol chosen for the serial boot port is the Serial Peripheral Interface (SPI), a standard to which a variety of EEPROMs on the market are built. The primary features of the SPI interface are minimum pin count, compliance with a standard protocol, the ability to self-start without requiring configuration, the ability to share one EEPROM among multiple devices, and the ability for a processor to directly read and write the EEPROM during field updates of the boot loader.

After the system has been reset, the boot interface sends a "read data" command out to the EEPROM, which responds by sending the boot data back to the system. The first few bytes contain a byte count of the remaining boot stream and a set of flags. The interface writes the remaining data into the shared memory, starting at address 0. Once the boot stream

is complete, the interface uses the flags to determine which processors to start, and issues a signal, via the global resources, to start this processor. Typically, this process is the first state of bootup. Once a processor has been started it will contact the host system through either the network cell bus or Host Port interfaces and will load the remaining boot data directly from the host.

The serial boot interface can be used in a system where a single EEPROM serves multiple systems. In this case, a pin signal indicates which system is to be the boot master. This master boots normally, while slave systems "snoop" this boot stream and load the data as if they had issued the command. In addition to normal post-reset boot, the boot interface can be triggered under software control. This can be used to reboot the system under software control or can be used by a boot master chip to reboot a slave. In the later case, the master chip simply issues the boot command to EEPROM and ignores the data being returned.

Finally, the interface can be controlled directly from any of the system's processors through the global resources. In this mode, any standard SPI command can be issued to the EEPROM and data can be read back by the processor. In addition, the processor can use this mode to reprogram the EEPROM.

The line time-division multiplexer (TDM) ports 514 are serial data I/O ports that allow many independent communication channels to share a single interface. These ports connect the system with front- end devices such as those that convert analog signals into bitstreams and vice versa. TDM protocol divides time into uniform quantities called frames. Each frame is further sub-divided into slots, each eight clocks long. Each channel is granted one slot per frame and can transfer one byte during that slot. Each port consists of one input and one output data channel. Each channel is bit-serial and runs asynchronously with respect to the other channels and the rest of the chip. Each serial data line is accompanied by a clock and frame pulse which allows the interface to distinguish bit and frame boundaries.

The system's TDM interfaces 514 maintain a table of buffer address pointers, one input and one output entry per channel. These point to circular I/O buffers in cluster or shared memory, one per channel. The addresses and sizes of these buffers are controlled by the OS running on the system's processors. As data arrives at the pins, the interface converts the serial stream into 32 bit words per channel and writes the data into the appropriate

channel input buffer, updating its pointers as necessary. At the same time, the interface automatically pulls data out of the appropriate output buffer and serializes it for transmission on the outgoing port. This process proceeds continuously as long as the interface is turned on.

Status information on the current state of the I/O process, including the address of the last value written or read is stored in the I/O buffers by the TDM interfaces. This information can be read by a processor in order to monitor the I/O process. In one embodiment, the system's TDM interfaces are designed to handle a wide variety of TDM protocols at frequencies ranging from 1-32 MHz and up to 128 channels per port.

The basic protocol for a TDM frame begins with a frame pulse, which in this example is coincident with the first data byte of the first channel's data. After the frame pulse, one byte of data is generated or accepted for each channel in turn (on a TX or RX TDM, respectively) until the end of the frame. Even though the design allows multiple channels to be processed, it is also possible to operate with just a single channel. Although a frame may have up to 256 channels, an individual TDM port can handle the data for 128 channels; it is capable, however, of ignoring the channels for which it is not responsible. This provides a mechanism by which multiple TDM ports can be connected to share a single TDM data stream.

The line side TDM port supports different modes. The frame pulse can be chosen to be active-high or active-low. The frame pulse can be chosen to update on a rising or falling clock edge. The data can be chosen to update on a rising or falling clock edge..

The frame pulse can occur coincident with the first data bit of each frame, or in its own time slot after the last data bit of a frame but before the first data bit of the next frame The data bytes in each time slot can be arranged MSB-first or LSB-first. The system's TDM ports can generate frame pulses or receive them from outside. The frame size can be adjusted from 1 to 256 channels. The system can source or receive the TDM clock

The TDM ports implement a scatter–gather capability. This lets each receive and each transmit channel have its own independent, contiguous data buffer in the cluster memories. The system's line side TDM ports scatter incoming data frames so that one byte is written into each channel's data buffer in the cluster memory of the cluster handling that channel. Similarly, they gather outgoing data frames by reading one byte from each channel's buffer in

the cluster memory of the cluster handling that channel. This capability lets the processes that will be producing and consuming data concern themselves with a single memory buffer and a single channel at a time.

The scatter/gather feature of the TDM port allows the clusters to handle the data from each channel in its own buffer area, independent of the other channels. Each channel has its own buffer area in cluster memory or shared SRAM, which is structured in a certain way. Each buffer area in a cluster memory is allocated to one channel. The last two bytes within the buffer are reserved for a pointer, and the rest is a circular storage "ring" buffer for data. As data arrives from the outside world, the TDM port writes it into the buffer, one byte at a time. Every time an eighth byte is written, the TDM port generates an extra write cycle to rewrite the pointer to indicate where in the buffer the last write has occurred. By this mechanism, the client process that is using the data can read the buffer at any time and determine, within a granularity of 8 bytes, what data has been written by the TDM port and is ready to be operated on.

The actual value stored in the pointer byte is not a complete address. Rather, it is an offset from the start of the buffer to the data byte that has just been written. If a cluster processor wishes to know how far along in the data buffer the TDM port has written, it must read the pointer and add the result to the start address of the buffer. This produces the absolute address in cluster memory of the last byte of data that was written before the pointer was last updated. The TDM port has pointers in a RAM for the "Next Address", "Buffer Start", and "Buffer Length" values for each channel. The configuration data and status information for the TDM ports is stored in the register space, where it is accessible to the processors by writing to and reading from shared resources.

The network cell bus is a protocol that is used to move data between the system and a network cell bridge that links multiple systems and the rest of the host system. The network cell bus protocol breaks data into cells, each of which contains a header that distinguishes which channel the data is from, what kind of channel it is, a byte-count of the data in the cell, and other information. The network cell bus protocol is independent of the physical transport layer. The network cell bus is implemented on top of a TDM- style serial port. Physically, the I/O connections are identical to the TDM port connections. However, the network cell

bus interfaces interpret the frame pulse signal as the beginning of a new cell, rather than a new frame, and the entire cell is dedicated to a single channel.

Like the TDM interfaces, the network cell bus interfaces maintain a table of pointers to circular input buffers in cluster and shared memory, one for each channel. As a new cell arrives at the input port, the interface reads the header and determines to which channel that cell belongs. Based on its internal table, the interface writes the cell, including the header, to the appropriate input buffer over the shared I/O bus. On the output side, the process is different. A cell transferred on the network cell bus is composed of two main components, a header and data, and these do not necessarily need to be constructed in the same location in memory. Rather than having a separate output buffer per channel, the shared memory and each cluster memory have a circular buffer of pointers and flags. The pointers point to the various components of the cell(s) that need to be transmitted. The network cell bus interface reads the pointer memory in one of the cluster memories (or shared memory), and starts transmitting the data indicated by the pointers. When the sequence is complete (indicated by a flag in the pointer memory), the interface proceeds to the next cluster/shared memory block and repeats the sequence.

As with the TDM ports, the network cell bus ports write status information into the I/O buffers in order to keep the OS informed on the status of the I/O process.

The system's host port 518 is a standard host-processor interface supporting both Intel and Motorola bus protocols. The host port interface connects through the same I/O arbiter as the other I/O interfaces, but is limited to addressing the shared SRAM. The host port interface operates in two modes: master and slave. In master mode, the MP can command the host port to transfer data, as a DMA or single-cycle access, to and from the shared SRAM and the external system. In slave mode, the host port maps read/write accesses from an external processor into reads and writes from the shared SRAM. The host port also provides a set of registers that can be read/written to by both the MP and the external host processor. These registers can be used for coordination. Host port status information is available in registers that can be read by the MP. The host port is asynchronous with the core clock, but may be synchronous or asynchronous with the external processor. It can use a 8 or 16-bit data bus and a 16-bit address bus and can transfer data at 17-20 Mbytes/sec.

The Shared Memory

In one embodiment, the system's shared memory system holds program instructions and data used by the system. In addition, because MP 600 does not directly access the cluster memories (101, 201, 301, 401, Fig. 4) shared memory system provides the memory available to the MP 600. In one embodiment, as illustrated in Fig. 6, the shared memory consists of three main components including on-chip shared SRAM memory configured as two independent banks 508, 510, a DRAM controller 520 for optional external DRAM 522, and a central arbiter 502 that controls the access to these memories. Different numbers of banks and other components may be used in other embodiments.

The on-chip shared memory banks 508, 510 hold actively used code and data that is shared between clusters 100, 200, 300 400, or between the clusters and the MP (600, Fig. 5). The shared memory 508, 510 provides relatively low-latency accessibility to this data. In one embodiment, the two SRAM banks 508, 510 of the shared memory consist of 192 KBytes each. Those of skill in the art will recognize that additional memory banks could be used, and the size of these banks may increase or decrease according to the application. In one embodiment, reads to the banks are 128 bits wide and writes are 32 bits wide with byte-enables. One read or one write is allowed per bank per cycle.

DRAM controller 520 connects the system to off-chip DRAM memory 522, which can be up to 64 Mbytes in size. The optional DRAM 522 provides a place to store large quantities of less-frequently used data or program code. Data can be accessed directly from the DRAM 522, or can be transferred to the shared memory 508, 510 or cluster memories (101, 201, 301, 401 Fig. 4) before use. Any cluster (104, 204, 304, 404, Fig. 4) memory bridge or the main cluster memory bridge (614, Fig. 5) can perform a direct memory transfer. Depending on the system configuration, the datapath to DRAM 522 can be either 16 or 32 bits wide.

The global shared memory (i.e. shared memory SRAM 508, 510 and external DRAM 522) is not cache coherent among the chip's processors. Coherency can be maintained in software. This can be accomplished in one embodiment through a global inter-processor lock and barrier synchronization mechanism described in more detail below.

As can be seen from the block diagram of Figure 16, the shared memory crossbar switch and arbiter is very similar to that used in the cluster to service access to the cluster memory. One of the differences is that there are more requestors than there are banks of memory in the one embodiment described in Figure 16. Nevertheless, multiple accesses can occur as long as the requests do not contend for the same bank. The function of the central arbiter (and crossbar switch 990) is to receive requests from the clusters (four processing clusters and one main cluster) 980 and the IO bridge associated with the Shared Memory and passes those requests to the Shared Memory banks 508, 510 or DRAM efficiently. This one embodiment of the Central Arbiter/Shared Program Memory structure is capable of sustaining up to 5.3 GB/s of read/write traffic at a clock rate of 166 MHz. Efficient access to shared memory is important for an architecture that is pooling memory. This embodiment of the Arbiter/Shared Program Memory structure ensures that as many processor engines as possible are working on the execution of program instructions,

The Central Arbiter/Shared Shared Memory system has three independent arbiters and associated transaction source multiplexers, along with six front-end interfaces 980 to manage the interaction between the Central Arbiter and the requesters 980. It also has an interface to the SDRAM Memory. The Central Arbiter manages the routing of requests from eleven requesters (five processors, five bridges, and the IO arbiter) to three resources (two blocks of Shared Memory and an SDRAM memory. It does so by managing a set of request queues, which are serviced by the resources as dictated by a hierarchical system of arbitration. Requests enter the Central Arbiter through a Front End block 980 (cluster, MP, or I/ O), where they are queued and dispatched to the Resource Arbiters. The Front End interfaces manage the various request queues (including an IO queue or processor shared Shared Memory/DRAM queue, a bridge Shared Memory queue, and bridge DRAM queue). To reduce routing congestion, the processor and bridge within a cluster share a single 128-byte return data bus (the I/O return data bus is 32 bytes). It is the responsibility of the Front End interface to ensure that requests do not generate contention on this shared return data bus, that is, only one resource can be returning data to a requester in a cluster at a time. Each Front End passes up to one Shared Memory request and one DRAM request to the Shared Memory/DRAM Resource Arbiters. The Resource Arbiters examine the various active requests, and can handle up to three of them concurrently. The Shared Memory access requests are routed straight to the Shared Memory blocks, while the DRAM requests are sent

time-multiplexed to the DRAM Controller Interface block where they are passed on to the DRAM Memory Controller for servicing.

Once a request has been satisfied by a resource, the data returns on the Return Data Bus corresponding to that resource and is then routed by the appropriate Front End to the initial requester.

A request enters the Central Arbiter through a Front End interface block, where it is queued for servicing. Processor requests are stored in a combined Shared Memory/DRAM queue. As the processor cannot issue more than one read transaction, the combined queue does not have a negative impact on read latencies (nor does it have any effect on writes). Bridge requests, however, are routed to separate Shared Memory and DRAM request queues, as the bridge can have multiple outstanding transactions. This structure allows Shared Memory requests to proceed unhindered by DRAM requests and yields the highest efficiency for bridge transfers.

Once queued, the processor and bridge DRAM requests are serviced round-robin by the First Level DRAM Arbiter. Requests are then propagated to the top-level DRAM Resource Arbiter. If the cluster wins arbitration at that level, the transaction information is transferred to a holding register in the cluster Front End and issued in a time-multiplexed fashion to the DRAM Controller Interface, which then passes it straight through to the SDRAM Memory Controller.

The holding register is provided so that the next transaction in the processor queue can be serviced on the subsequent clock cycle, rather than holding it off during time-multiplexed transfer. Similarly, requests come to the Return Wire Arbiter from the processor and bridge Shared Memory queues, as well as the DRAM Memory Controller (via the top-level DRAM Resource Arbiter). The DRAM Memory controller notifies each cluster in advance that return data destined for that cluster will be available shortly. When no DRAM data is scheduled for return, the arbiter services Shared Memory requests in round-robin fashion. However, when notified that data is returning from the DRAM Memory controller, the returning data is granted priority. Thus, Shared Memory transactions arbitrate for the cluster Return Data Bus at request time, while DRAM transactions arbitrate for it on data return. Upon winning local arbitration, Shared Memory requests are propagated to the top-

level Shared Memory Resource Arbiters. If the cluster wins arbitration at that level, the transaction information is multiplexed through to the appropriate Shared Memory block, and the corresponding cluster request queue is updated.

Pending read requests and DRAM data returns are logged by the Return Tracking Queue. This block pipelines the appropriate information to allow it to drive the correct select lines for the return data multiplexer and generate read acknowledge signals for the correct requester. The IO Front End manages a single queue containing Shared Memory access requests – no local arbitration is required. The Return Tracking Queue and data return logic tracks and aligns read requests to the appropriate word lane.

Every request enters the Central Arbiter and is routed to the appropriate request queue. In order to simplify both the control logic and interconnect, these are implemented as split address and data fifos. This means up to two transactions of any type may be queued at a time – a depth that enables requesters to stream requests continuously, barring contention. Because neither the processor nor the IO requesters generate burst write transactions, their data fifos are actually matched to the depth of the address fifos – two entries deep.

The control logic is responsible for fifo pointer and slot management, requester handshaking, and propagation of active requests to the cluster Return Wire Arbiter and First Level DRAM Arbiter. Each of the three Resource Arbiters examines the various active requests propagated by the Front End interfaces. Because IO requests are not throttled and have to be handled with a deterministic latency, the Shared Memory Resource Arbiters grants priority access to all IO transactions. For non-IO transactions, arbitration is on a purely round-robin basis – there is no state maintained as to the order in which transactions were received. Upon winning arbitration, Shared Memory requests are routed through the resource arbiter transaction multiplexer, straight to the Shared Memory blocks. DRAM requests are similarly routed, but are sent time-multiplexed by the DRAM Controller Interface to the SDRAM Memory Controller.

WHAT IS CLAIMED IS:

1. A multi-channel processing system comprising:

a plurality of processing engines, said processing engines sharing a data memory and a program memory; and

wherein said program memory retains a single copy of all of the programming instructions to be executed by said engines.

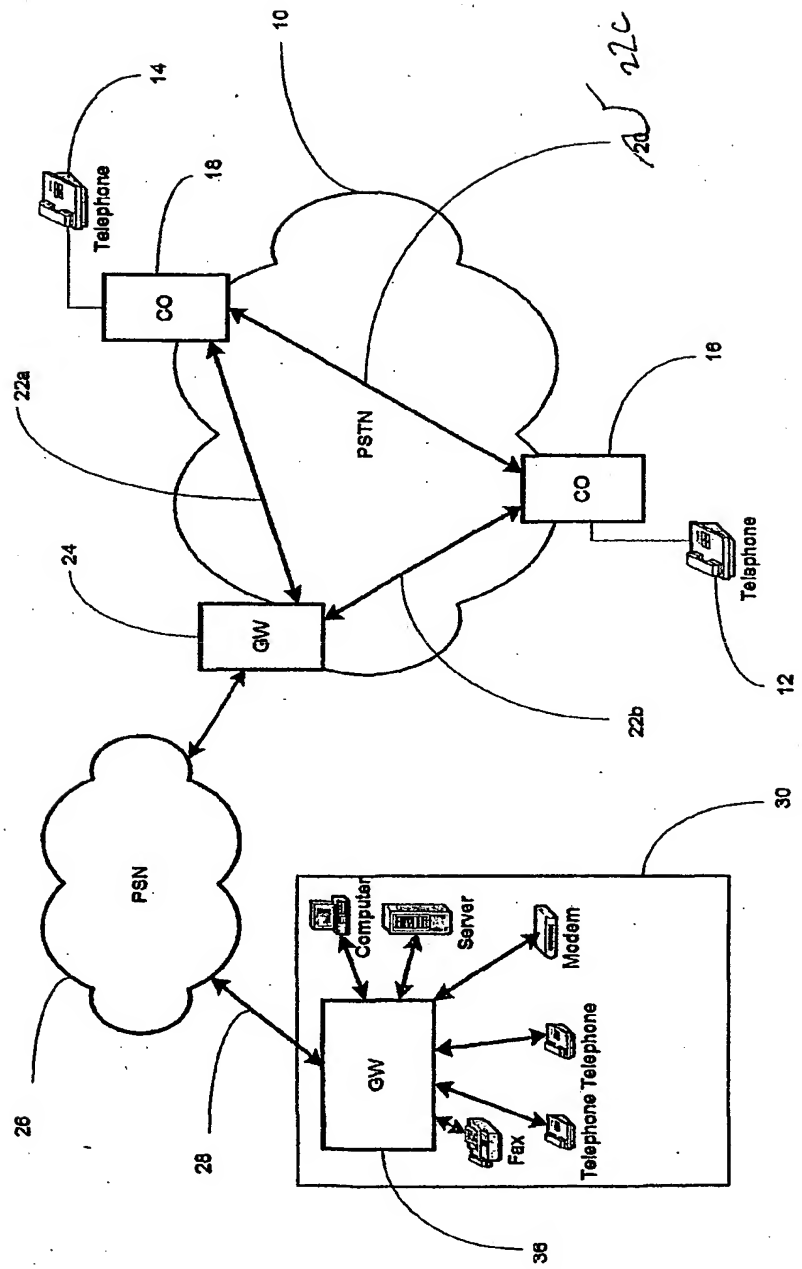

10
22c
14
Telephone
18
CO
22a
PSTN
16
CO
24
GW
Telephone
12
22b
PSN
30
Computer
Server
Modem
GW
Telephone Telephone
Fax
26
28
36

FIGURE 1

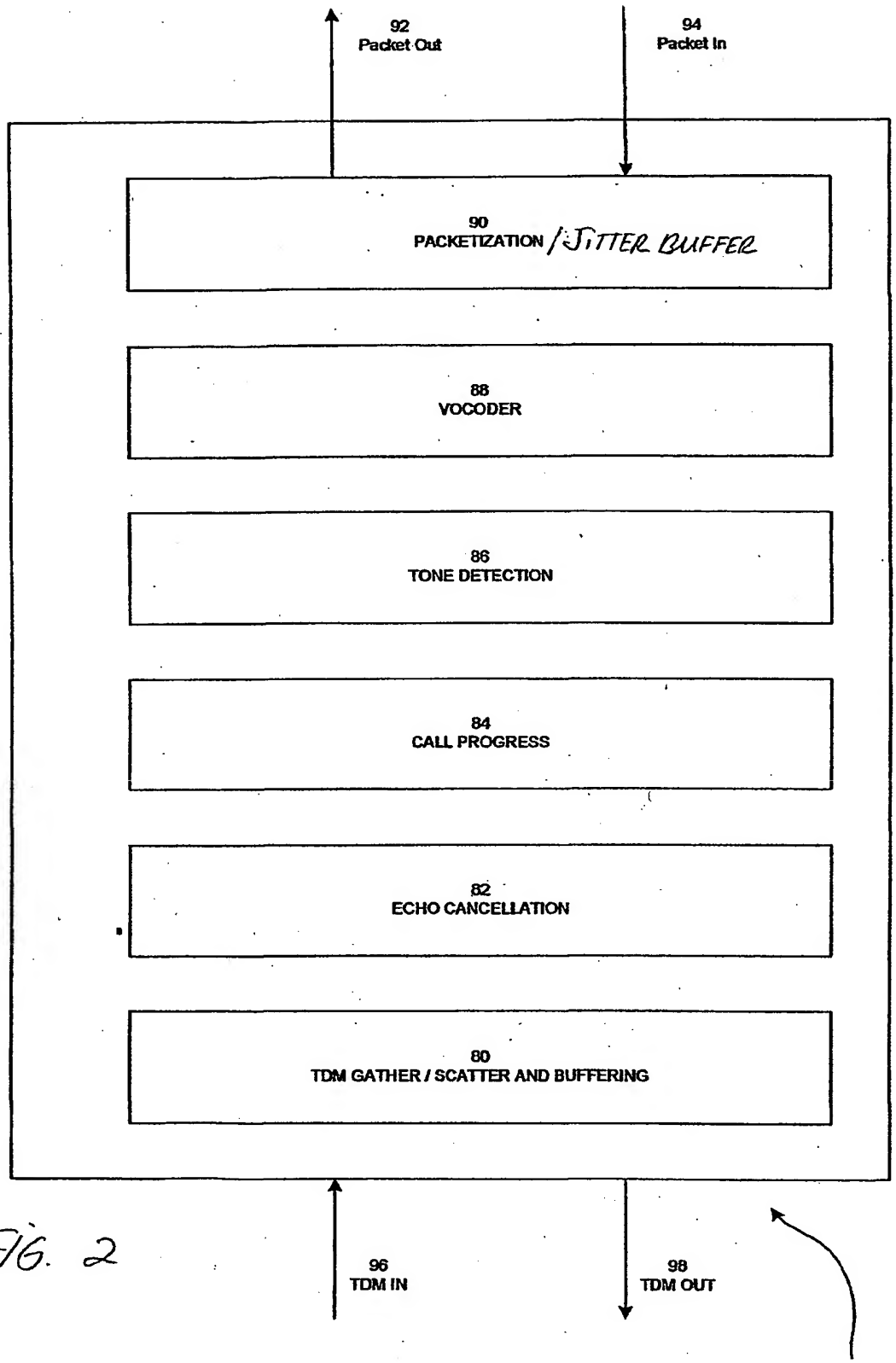

92
Packet Out
94
Packet In
90
PACKETIZATION / JITTER BUFFER
88
VOCODER
86
TONE DETECTION
84
CALL PROGRESS
82
ECHO CANCELLATION
80
TDM GATHER / SCATTER AND BUFFERING
96
TDM IN
98
TDM OUT

FIG. 2

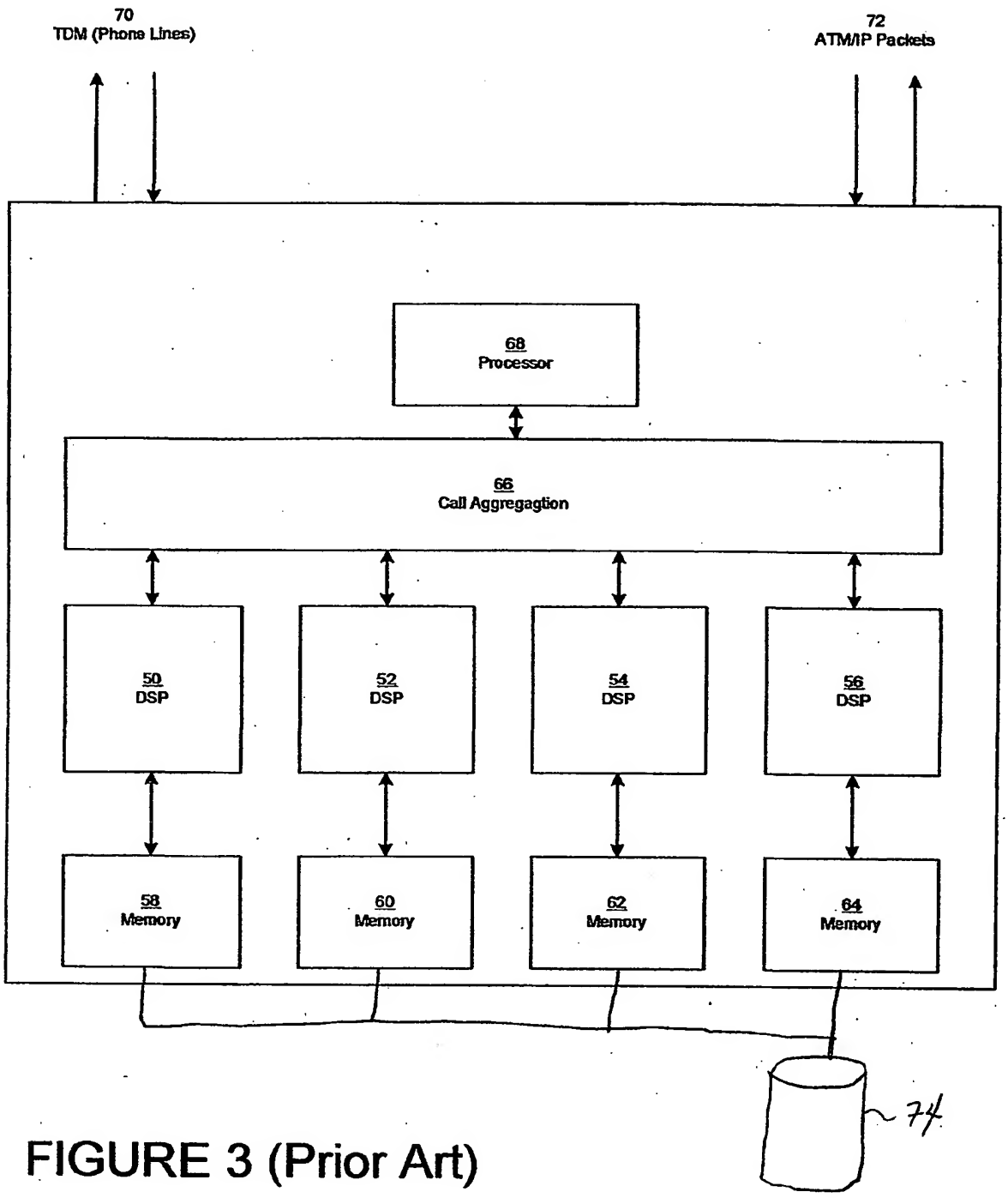


FIGURE 3 (Prior Art)

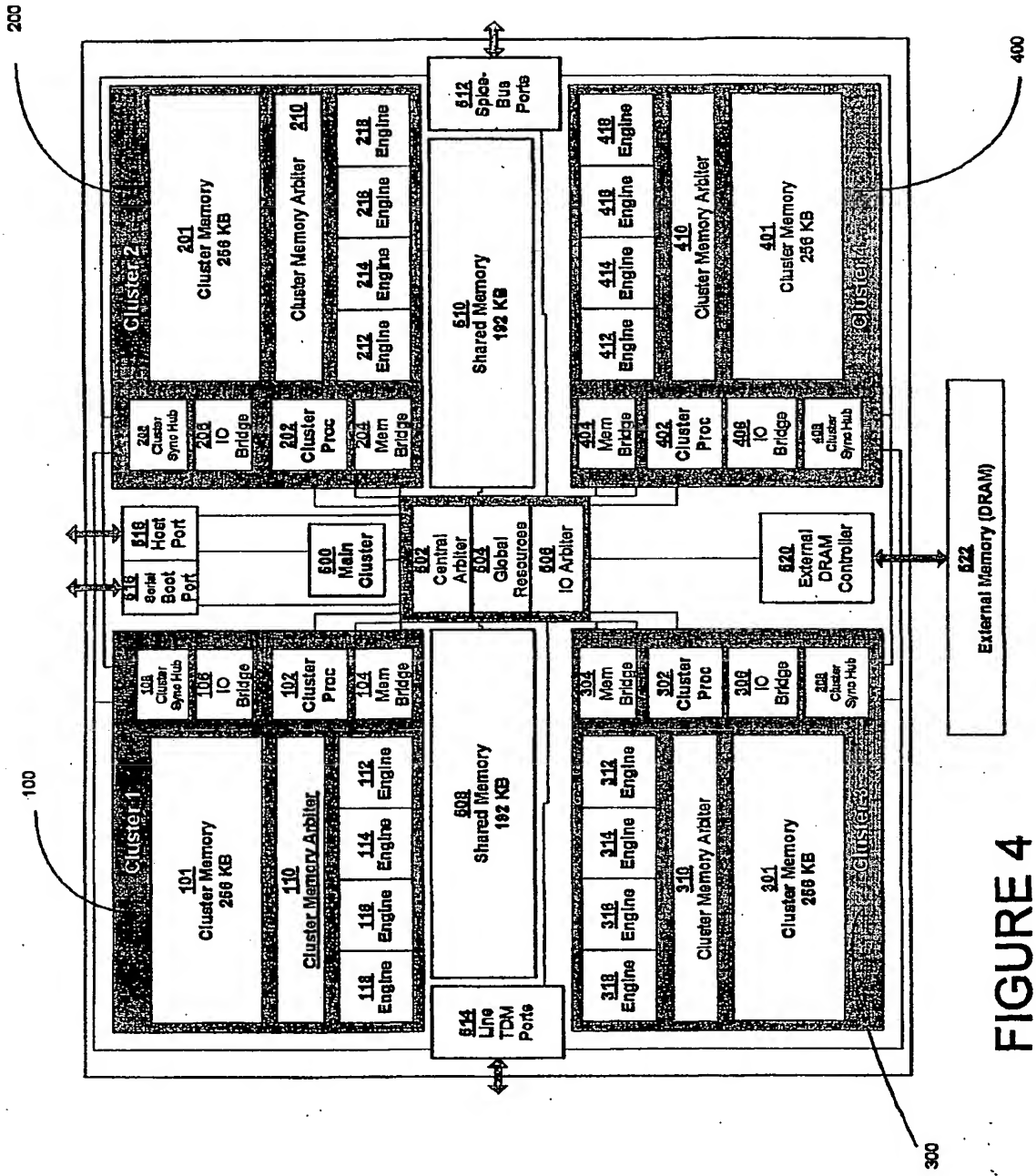

200
100
300
400
Cluster 1
101 Cluster Memory 256 KB
110 Cluster Memory Arbiter
118 Engine
116 Engine
114 Engine
112 Engine
108 Cluster Sync Hub
106 IO Bridge
102 Cluster Proc
104 Mem Bridge
608 Shared Memory 192 KB
614 Line TDM Ports
Cluster 2
201 Cluster Memory 256 KB
210 Cluster Memory Arbiter
218 Engine
216 Engine
214 Engine
212 Engine
208 Cluster Sync Hub
206 IO Bridge
202 Cluster Proc
204 Mem Bridge
610 Shared Memory 192 KB
612 Spica-Bus Ports
Cluster 3
301 Cluster Memory 256 KB
310 Cluster Memory Arbiter
318 Engine
316 Engine
314 Engine
312 Engine
308 Cluster Sync Hub
306 IO Bridge
302 Cluster Proc
304 Mem Bridge
Cluster 4
401 Cluster Memory 256 KB
410 Cluster Memory Arbiter
418 Engine
416 Engine
414 Engine
412 Engine
408 Cluster Sync Hub
406 IO Bridge
402 Cluster Proc
404 Mem Bridge
618 Host Port
616 Serial Boot Port
600 Main Cluster
602 Central Arbiter
604 Global Resources
606 IO Arbiter
620 External DRAM Controller
622 External Memory (DRAM)

FIGURE 4

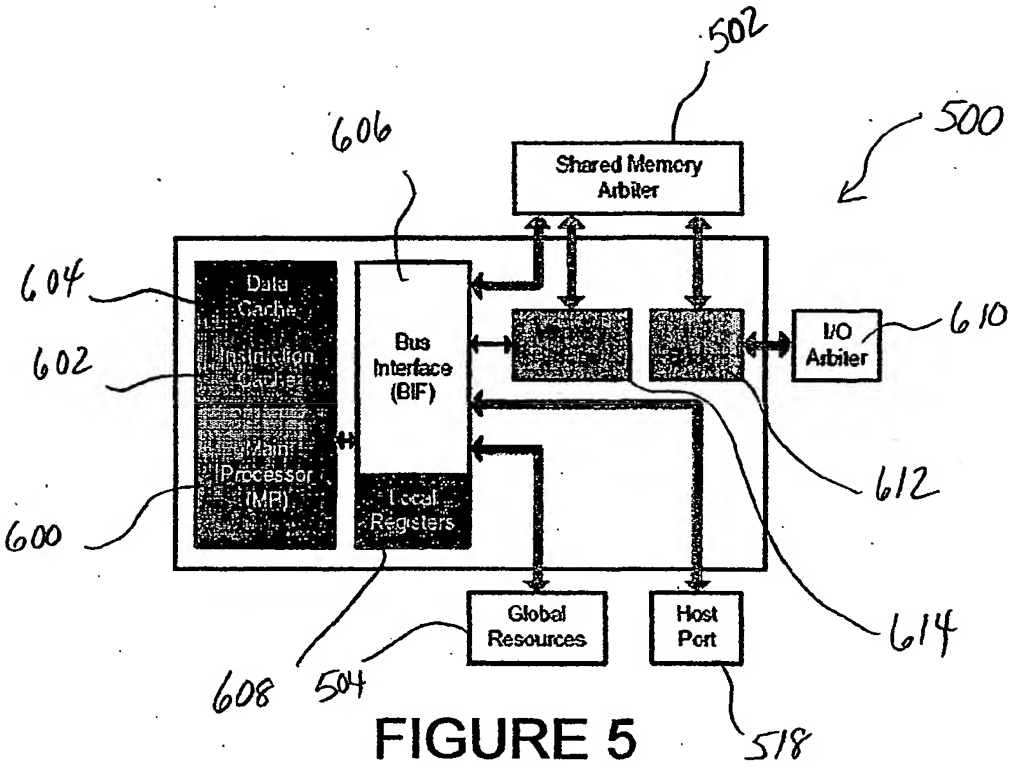

502
500
606
Shared Memory Arbiter
604
Data Cache
602
Instruction Cache
Main Processor (MP)
600
Bus Interface (BIF)
Local Registers
I/O Arbiter
610
612
614
608
504
Global Resources
Host Port
518

FIGURE 5

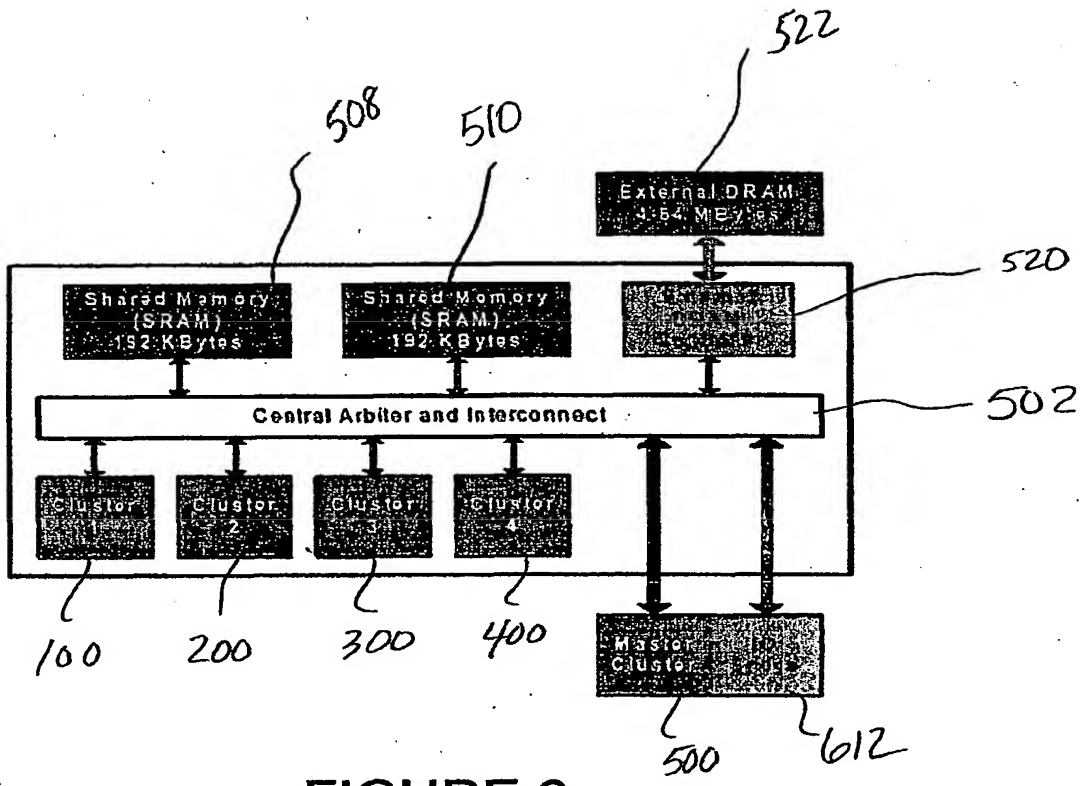

522
508
510
External DRAM
4-64 MBytes
520
Shared Memory
(SRAM)
192 KBytes
Shared Memory
(SRAM)
192 KBytes
Central Arbiter and Interconnect
502
Cluster
1
Cluster
2
Cluster
3
Cluster
4
100
200
300
400
Master
Cluster
500
612

FIGURE 6

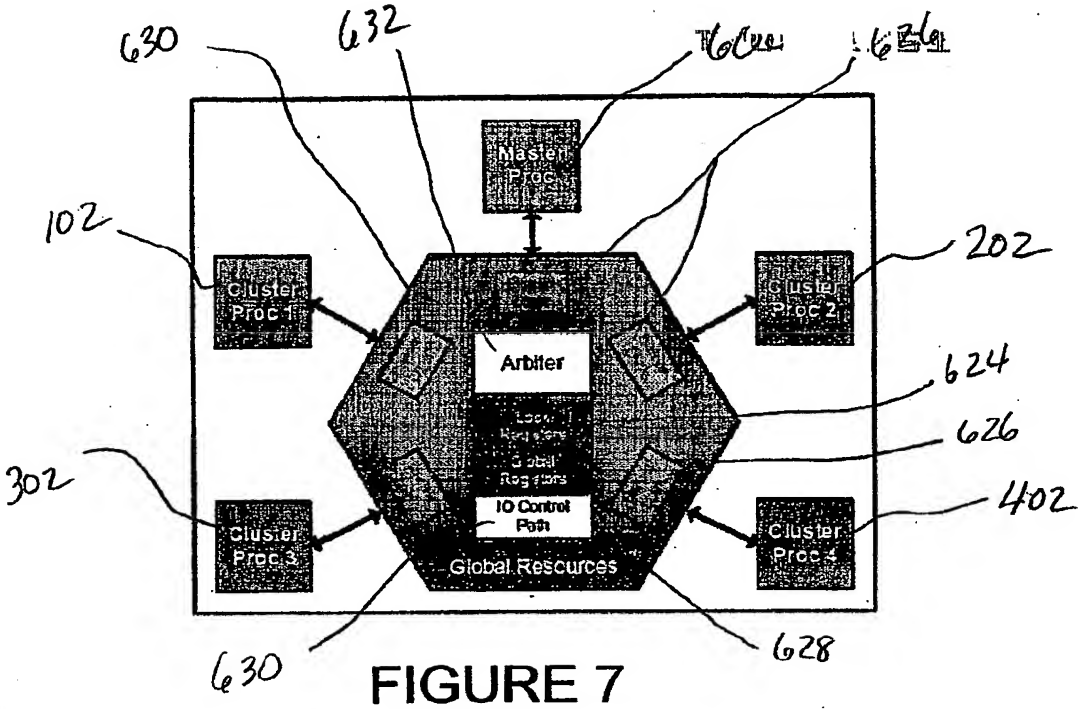


FIGURE 7

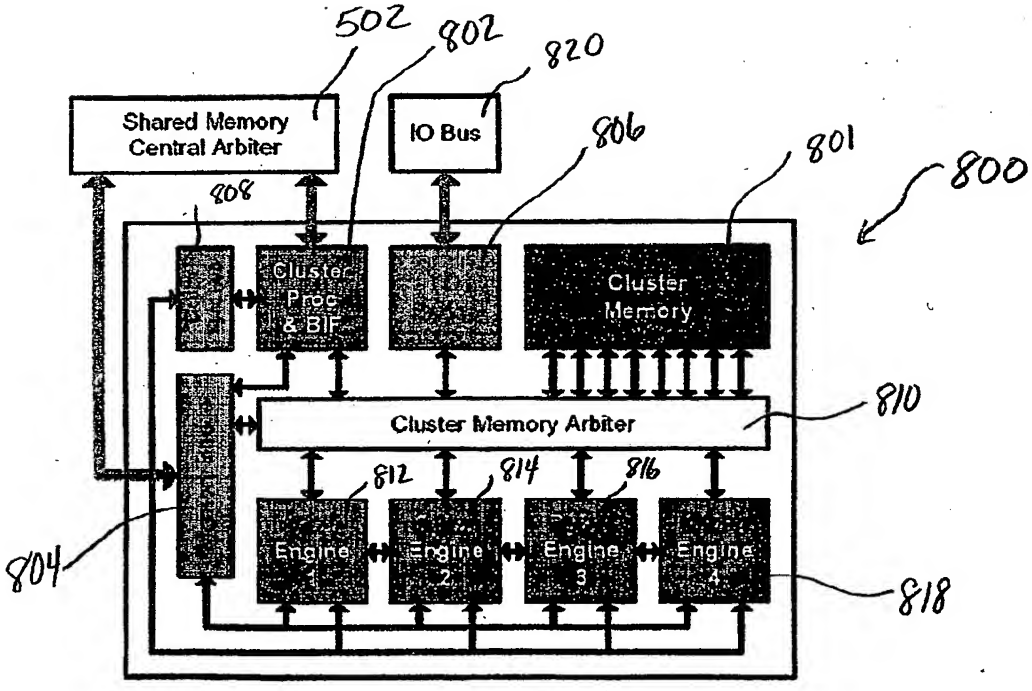


FIGURE 8

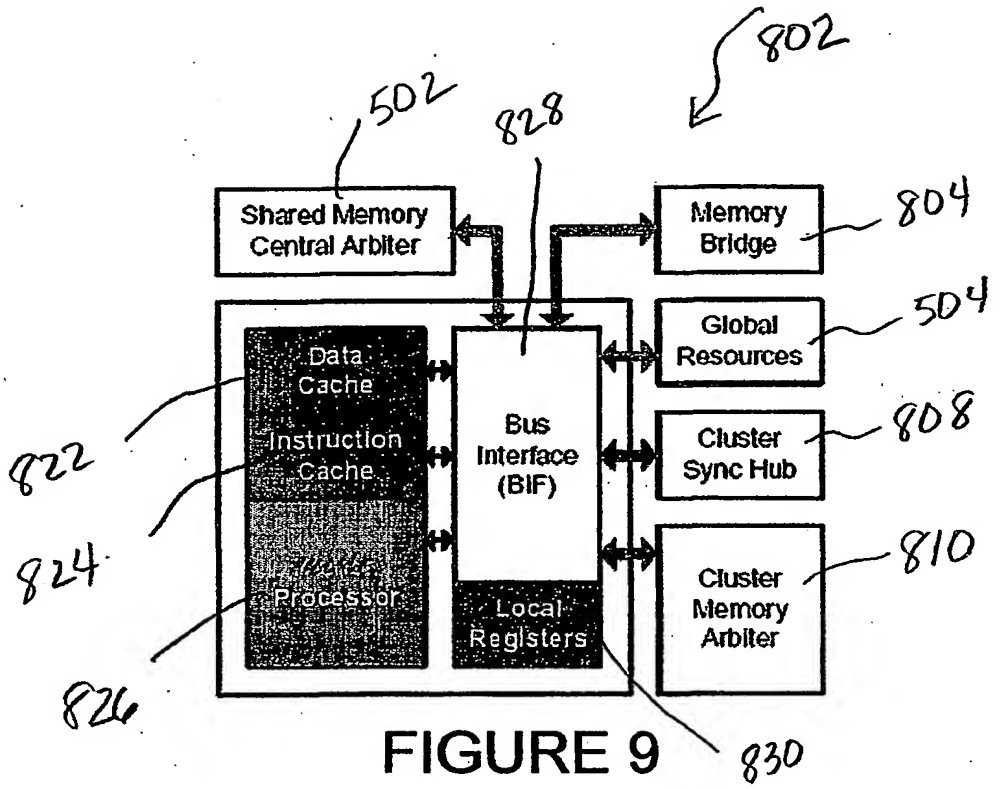

802
502
828
Shared Memory Central Arbiter
Memory Bridge
804
Global Resources
504
Data Cache
Instruction Cache
Processor
Bus Interface (BIF)
Local Registers
Cluster Sync Hub
808
Cluster Memory Arbiter
810
822
824
826
830

FIGURE 9

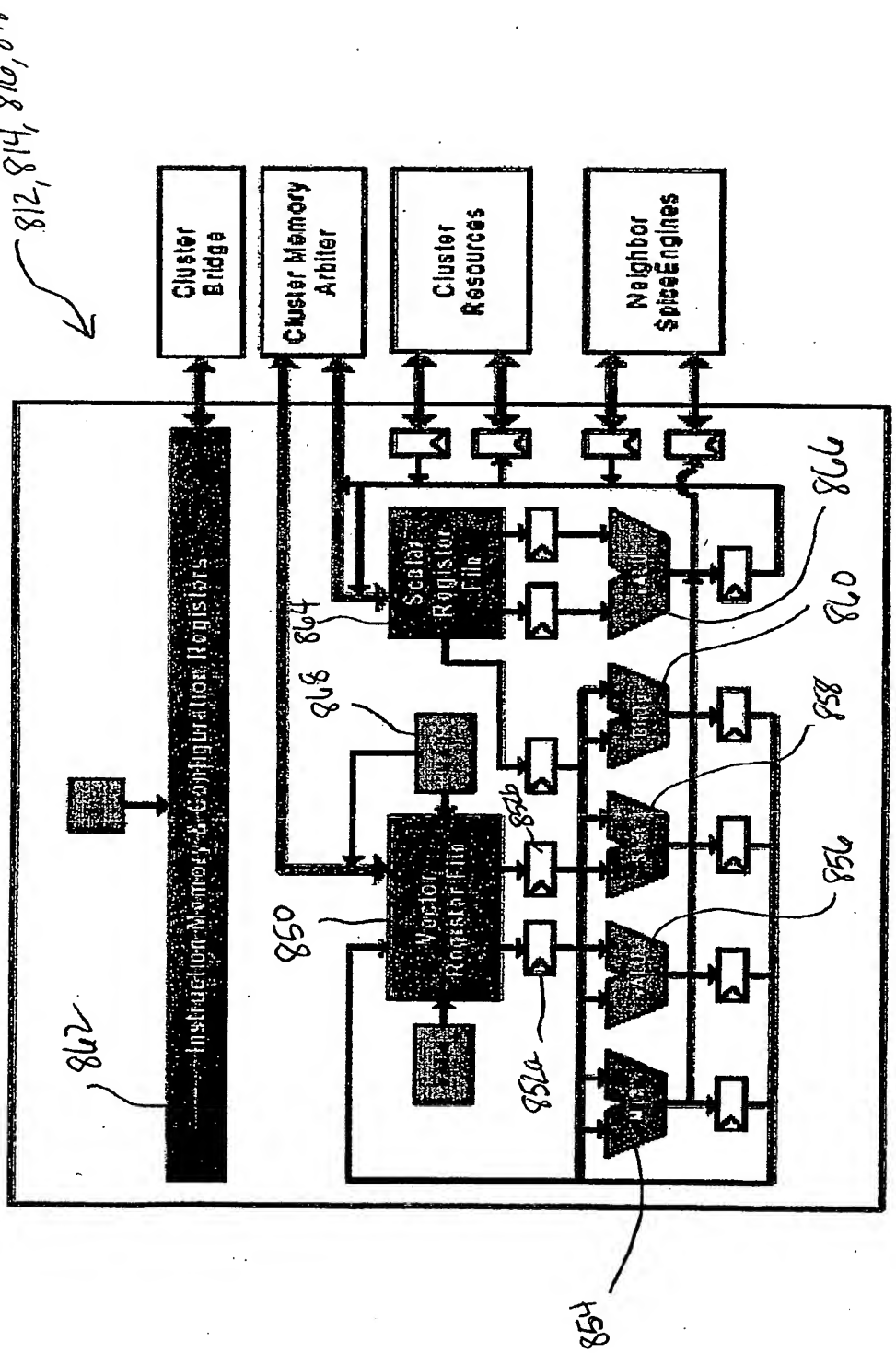

812, 814, 816, 818
Cluster Bridge
Cluster Memory Arbiter
Cluster Resources
Neighbor SpiceEngines
Instruction Memory & Configuration Registers
Scalar Register File
Vector Register File
ALU
ALU
ALU
ALU
ALU
812
850
852
854
856
858
860
862
864
866
868

FIGURE 10

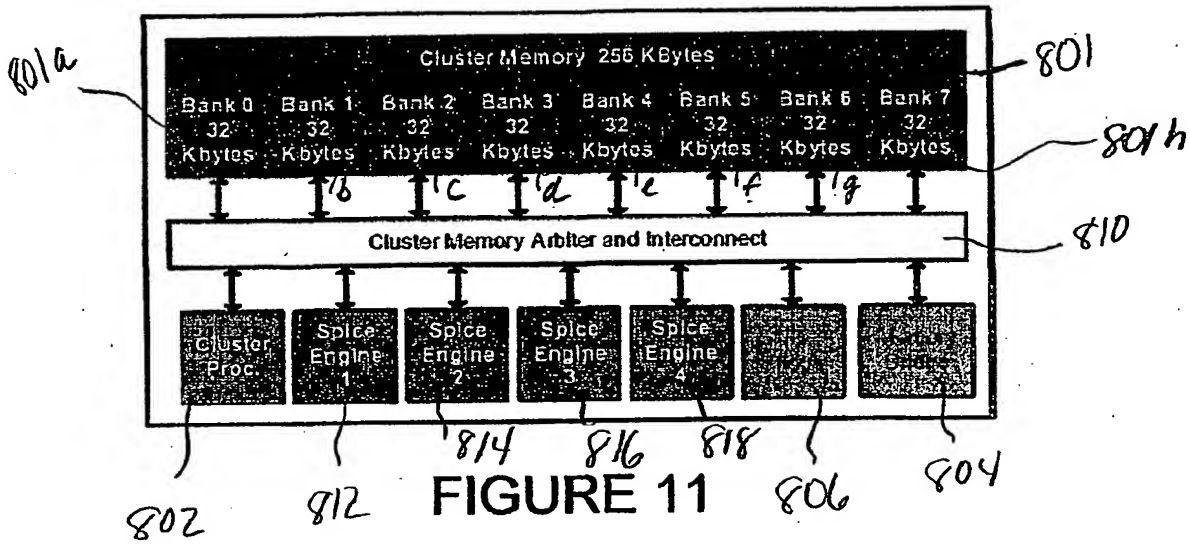


FIGURE 11

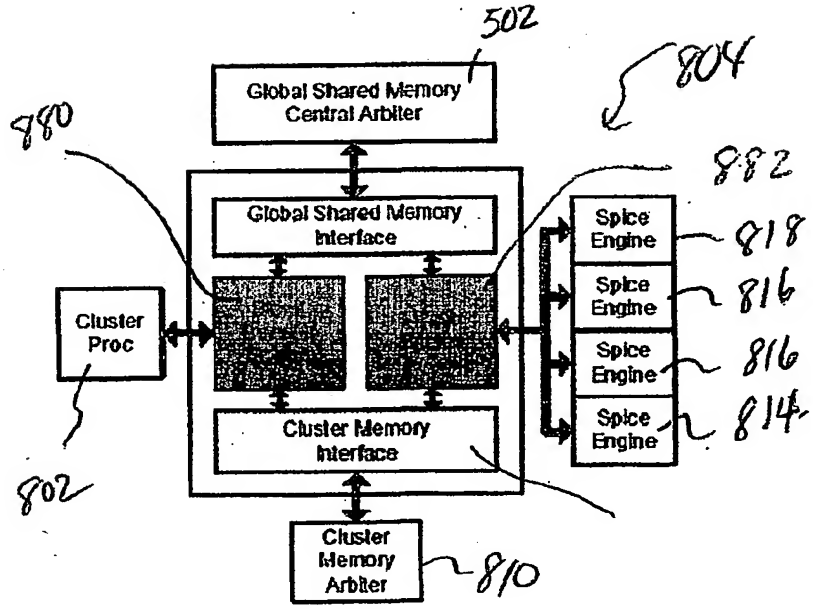


Figure 12

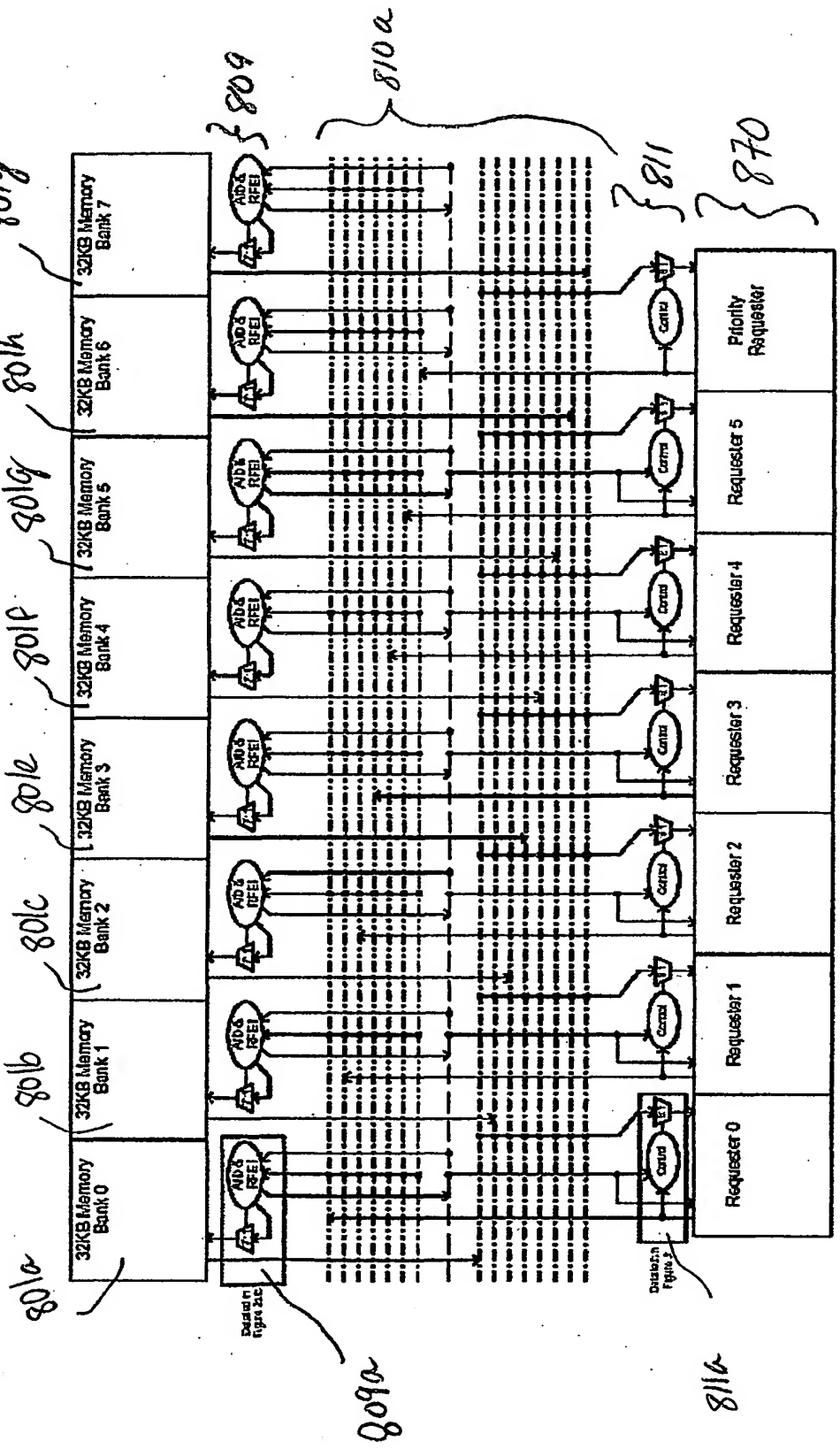

32KB Memory Bank 0
32KB Memory Bank 1
32KB Memory Bank 2
32KB Memory Bank 3
32KB Memory Bank 4
32KB Memory Bank 5
32KB Memory Bank 6
32KB Memory Bank 7
Requester 0
Requester 1
Requester 2
Requester 3
Requester 4
Requester 5
Priority Requester
Key:

FIGURE 13

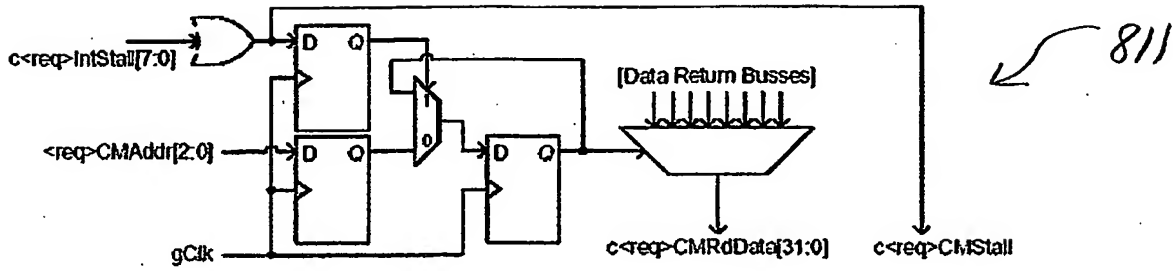


FIGURE 14a

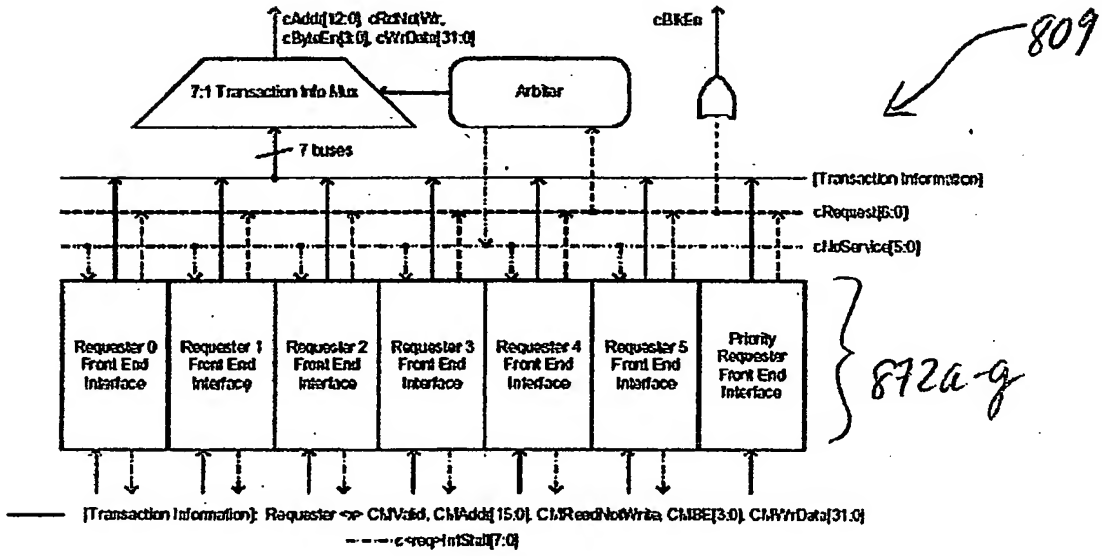


FIGURE 14b

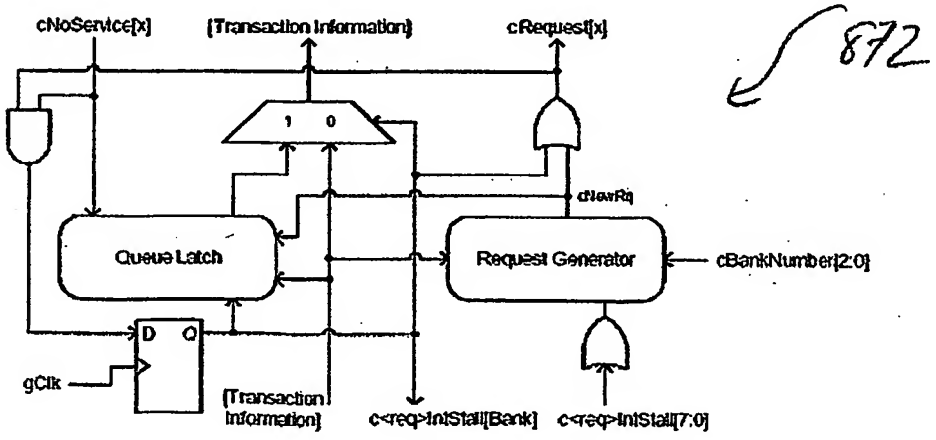


FIGURE 14c

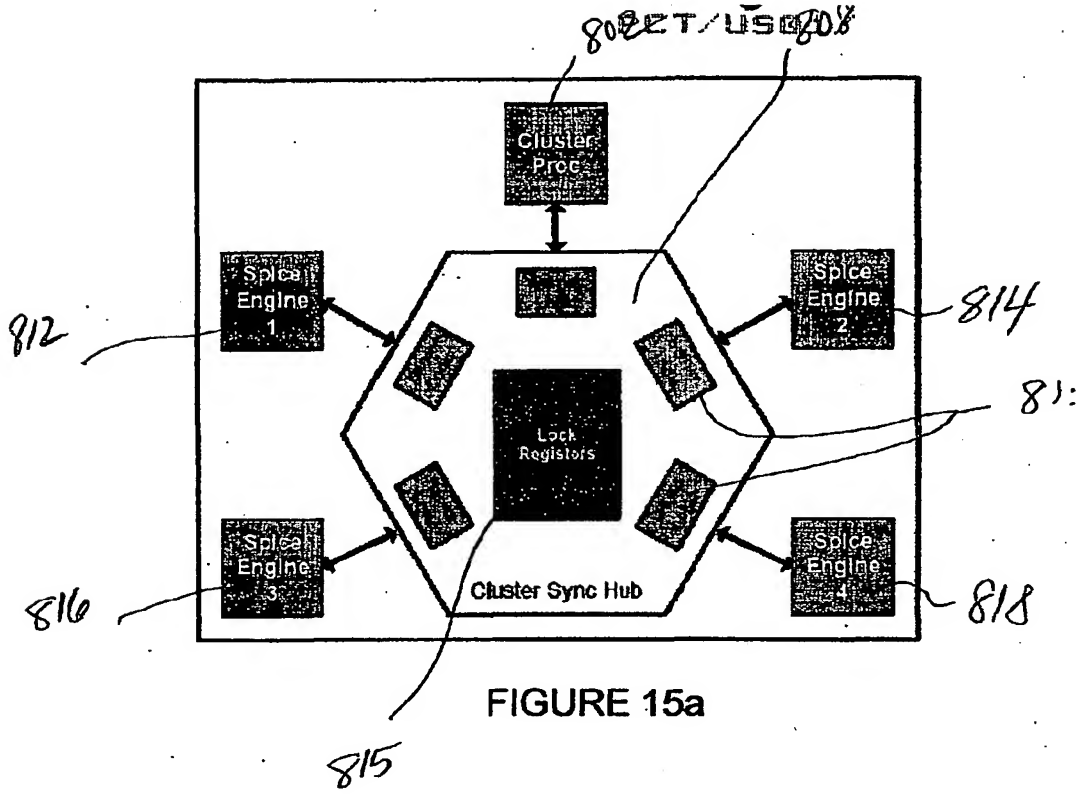


FIGURE 15a

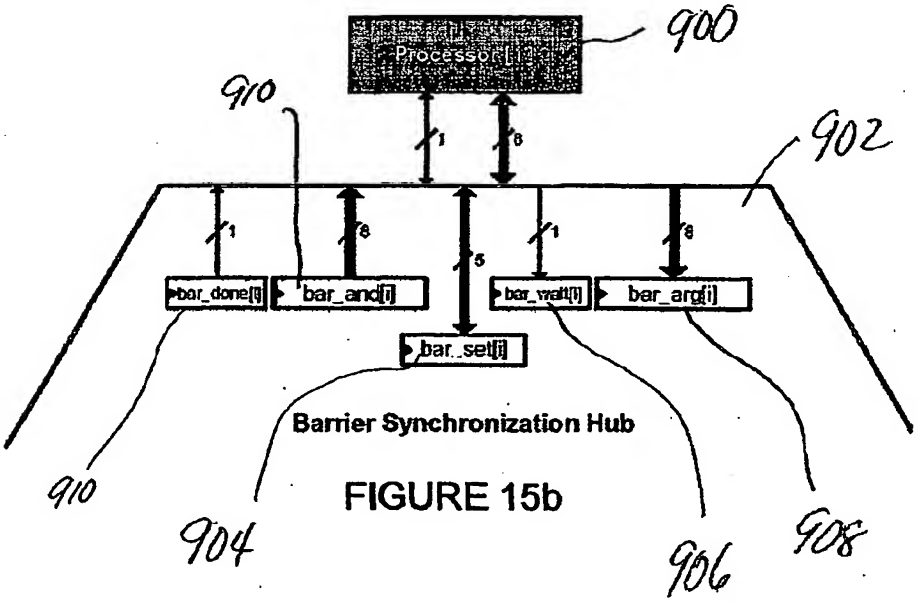


FIGURE 15b

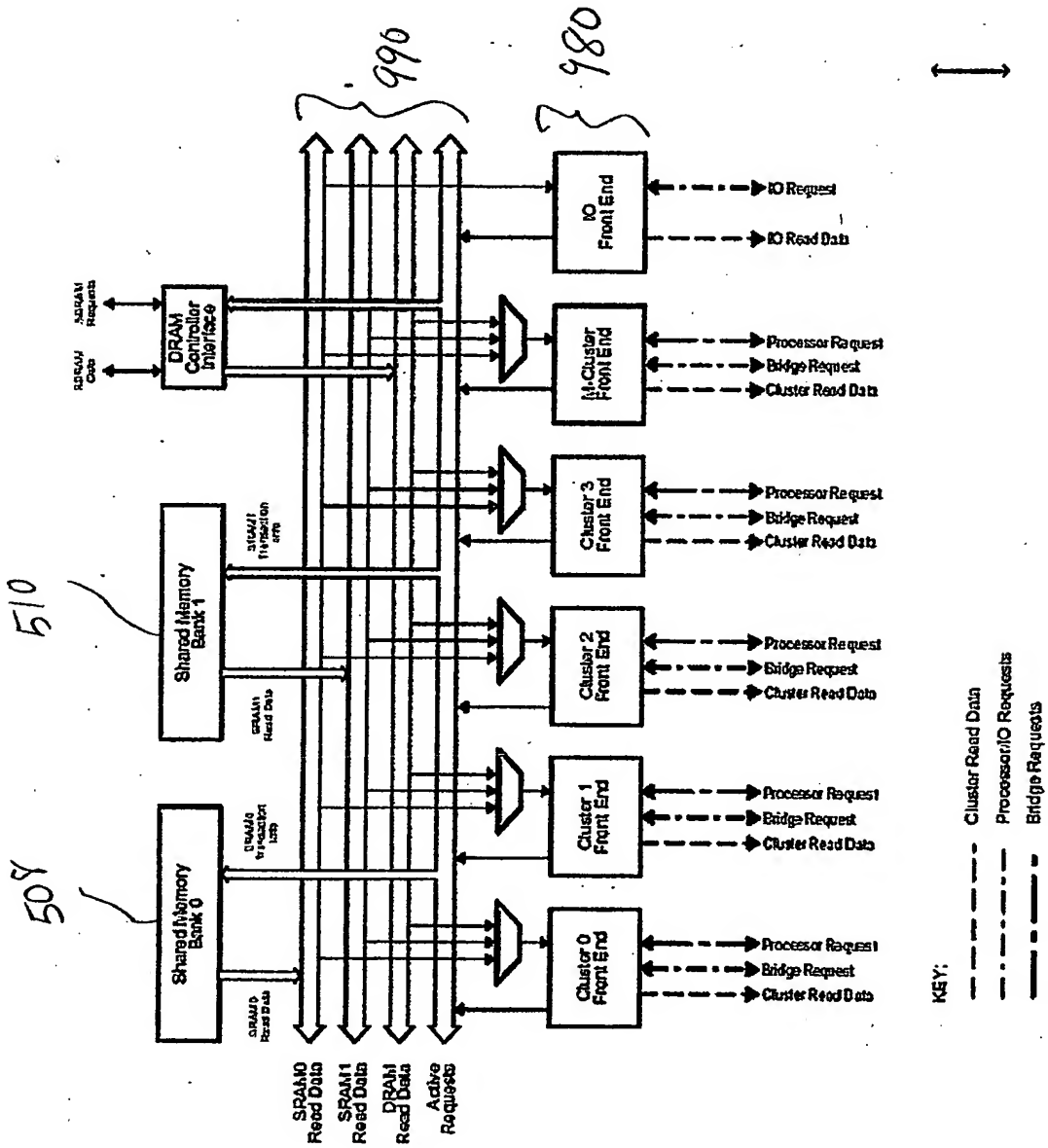

990
980
510
505
DRAM Controller Interface
Shared Memory Bank 1
Shared Memory Bank 0
IO Front End
M-Cluster Front End
Cluster 3 Front End
Cluster 2 Front End
Cluster 1 Front End
Cluster 0 Front End
IO Request
IO Read Data
Processor Request
Bridge Request
Cluster Read Data
SRAM0 Read Data
SRAM1 Read Data
DRAM Read Data
Active Requests
KEY:
Cluster Read Data
Processor/IO Requests
Bridge Requests

FIGURE 16

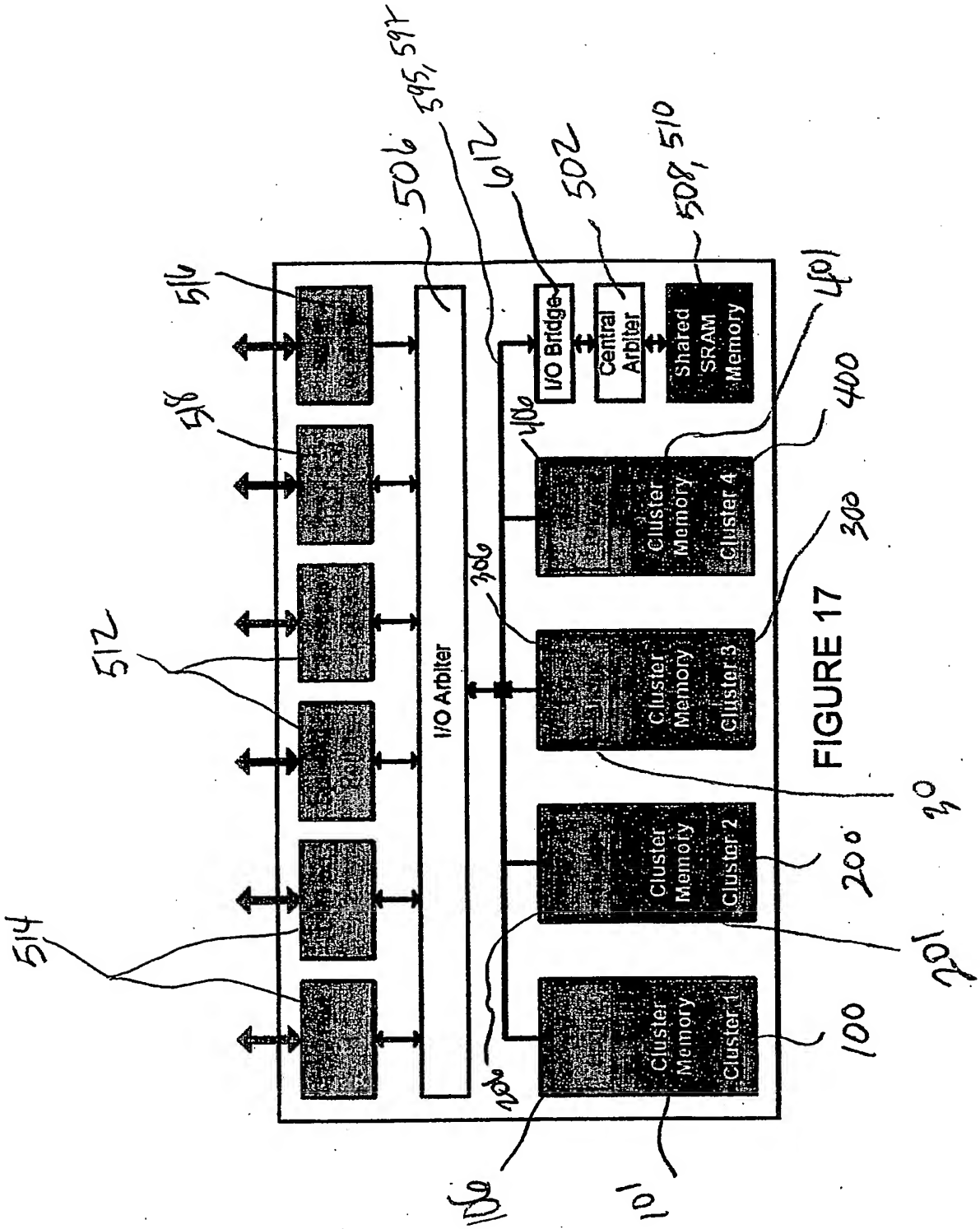

I/O Arbiter
I/O Bridge
Central Arbiter
Shared SRAM Memory
Cluster Memory
Cluster 1
Cluster 2
Cluster 3
Cluster 4
506
595, 597
502
508, 510
514
518
512
516
401
400
300
200
100
101

FIGURE 17

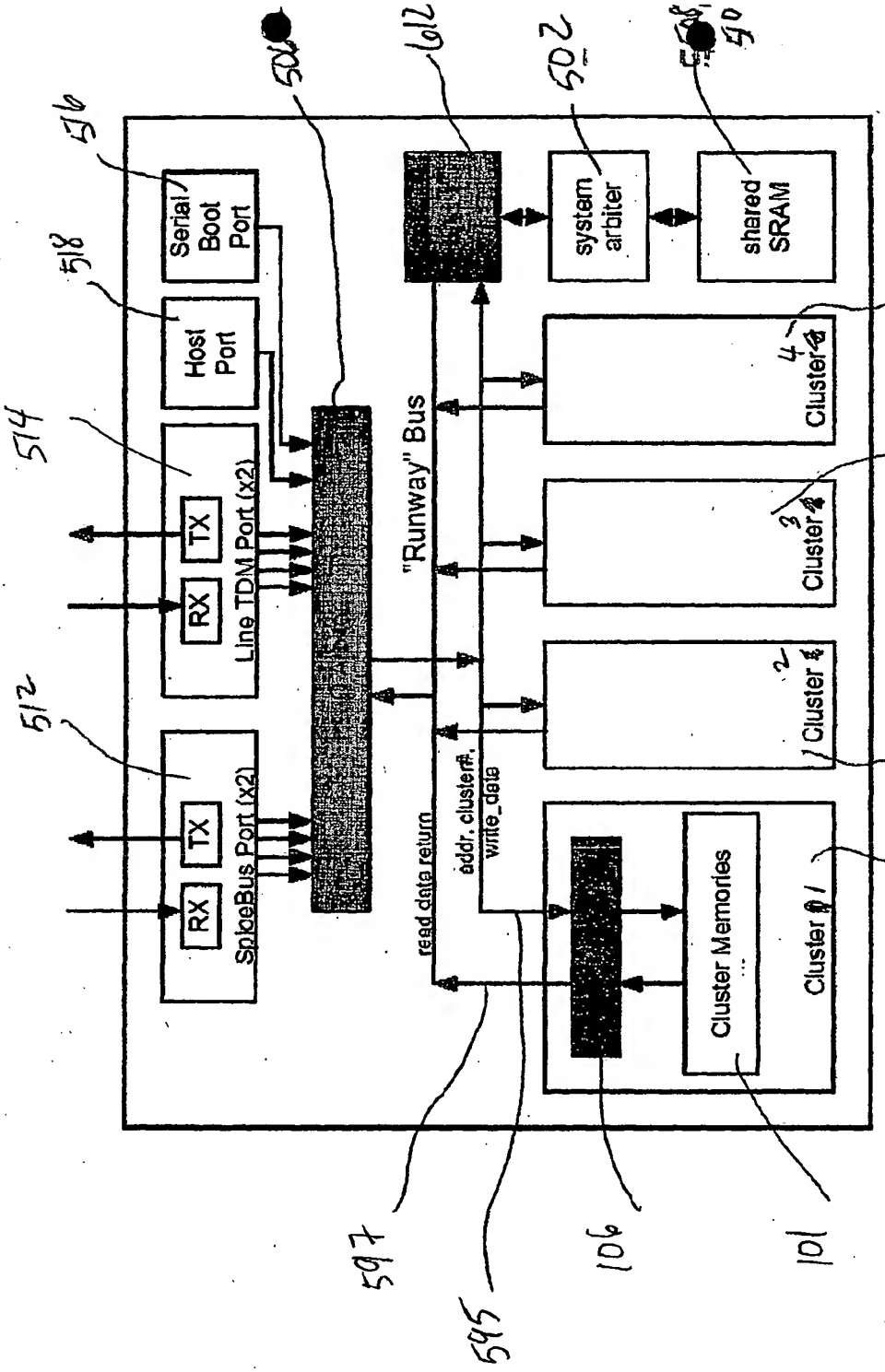

Serial Boot Port
Host Port
Line TDM Port (x2)
TX
RX
SpiceBus Port (x2)
TX
RX
"Runway" Bus
system arbiter
shared SRAM
Cluster 4
Cluster 3
Cluster 2
Cluster 1
Cluster Memories
read data return
addr, cluster#, write_data
516
518
514
512
502
612
504
508
400
300
200
100
597
595
106
101

FIGURE 18

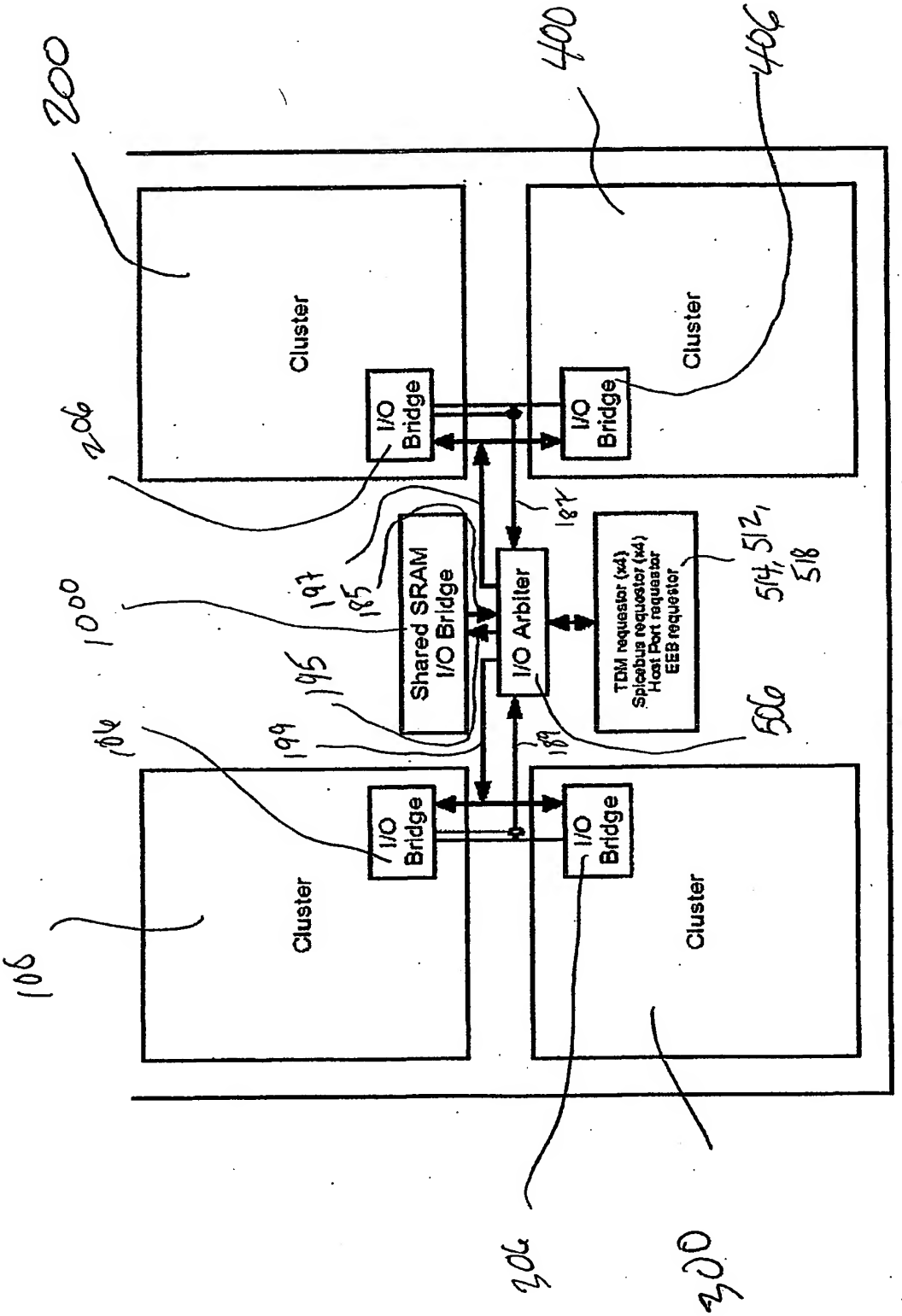

200
400
406
Cluster
Cluster
I/O Bridge
I/O Bridge
206
187
197
185
1000
Shared SRAM I/O Bridge
I/O Arbiter
TDM requestor (x4)
Splicebus requestor (x4)
Host Port requestor
EEB requestor
514, 512, 518
195
199
189
506
104
I/O Bridge
I/O Bridge
Cluster
Cluster
106
204
300

FIGURE 19